Kernel Methods

Prof. Nicholas Zabaras
Center for Informatics and Computational Science
https://cics.nd.edu/
University of Notre Dame
Notre Dame, Indiana, USA

Email: nzabaras@gmail.com
URL: https://www.zabaras.com/

March 22, 2019

Contents

Following closely:
Bishop CM, *Pattern Recognition and Machine Learning*, Springer, 2006 (Chapter 6)
K. Murphy, Machine Learning: A probabilistic Approach (Chapter 14)

Training Data: Keep or Discard?

- We have considered linear parametric methods e.g. for regression.
- The form of the mapping $y(\boldsymbol{x}, \boldsymbol{w})$ from input $\boldsymbol{x}$ to output y is governed by $\boldsymbol{w}$.
- During the learning phase, the training data is used either
  - to obtain a point estimate of $\boldsymbol{w}$ or
  - posterior distribution $p(\boldsymbol{w}|\boldsymbol{x}, \boldsymbol{t})$.
- The training data is then discarded, and predictions for new inputs are based purely on the learned parameter vector $\boldsymbol{w}$.
- The same approach is used in nonlinear parametric models including neural networks.

Training Data: Keep or Discard?

- There are pattern recognition techniques, in which the training data points are kept and used for prediction.
  - Parzen probability density model: set of kernel functions centered on training data points.
  - Nearest neighbors technique: assigning to each new test vector the same label as the closest example from the training set.
- These are memory-based methods: storing the training set in order to make predictions for future data points.
- They require a metric to measure the similarity of any two vectors in input space.

Kernel Functions

Kernel Function

- We define a kernel function to be a real-valued function of two arguments, $k(\boldsymbol{x}, \boldsymbol{x}') \in \mathbb{R}$, for $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X}$.

- Typically the function is symmetric (i.e., $k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$), and non-negative (i.e., $k(\boldsymbol{x}, \boldsymbol{x}') \geq 0$), so it can be interpreted as a measure of similarity (but this is not needed).

- Kernel methods are generally fast to train but slow at making predictions for test data points.

Kernel Methods and Kernel Trick

- Many linear parametric models can be recast into an equivalent 'dual representation'
  - **Prediction is based on linear combinations of a kernel function evaluated at the training data points**
- For models which are based on a fixed nonlinear feature space mapping $\phi(\boldsymbol{x})$, the kernel function is given by

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$$

- Using the identity mapping for the feature space we obtain the linear kernel:

$$\phi(\boldsymbol{x}) = \boldsymbol{x}, \quad k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$$

- Kernel trick: **if we have an algorithm formulated such that x enters only in the form of scalar products, then we can replace that scalar product with some other choice of kernel.**

- Aizerman, M. A., E. M. Braverman, and L. I. Rozonoer (1964). The probability problem of pattern recognition learning and the method of potential functions. *Automation and Remote Control* **25**, 1175–1190.
- Boser, B. E., I. M. Guyon, and V. N. Vapnik (1992). A training algorithm for optimal margin classifiers. In D. Haussler (Ed.), *Proceedings Fifth Annual Workshop on Computational Learning Theory (COLT)*, pp. 144–152. ACM.Scholkopf *et al.*,1998).

Types of Kernel Functions

- For models based on feature space mapping $\phi(\boldsymbol{x})$:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$$

- Its a symmetric function:

$$k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x}', \boldsymbol{x})$$

- Linear kernel:

$$\phi(\boldsymbol{x}) = \boldsymbol{x}, \quad k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$$

- Stationary kernel: invariant to translations in $\boldsymbol{x}$

$$k(\boldsymbol{x}, \boldsymbol{x}') = k(\boldsymbol{x} - \boldsymbol{x}')$$

- Homogeneous kernel (radial basis functions):

$$k(\boldsymbol{x}, \boldsymbol{x}') = k(\|\boldsymbol{x} - \boldsymbol{x}'\|)$$

Kernel Substitution Applications

- Algorithms expressed in terms of scalar products can be reformulated using kernel substitution (kernel trick). Applications include:
  - Non-linear PCA,
  - Nearest-neighbor classifiers,
  - Kernel Fisher discriminant, etc.

- Schölkopf, B., A. Smola, and K.-R. Müller (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation* **10**(5), 1299–1319.
- Mika, S., G. Rätsch, J. Weston, and B. Schölkopf (1999). Fisher discriminant analysis with kernels. In Y. H. Hu, J. Larsen, E. Wilson, and S. Douglas (Eds.), *Neural Networks for Signal Processing IX*, pp. 41–48. IEEE.
- Roth, V. and V. Steinhage (2000). Nonlinear discriminant analysis using kernel functions. In S. A. Solla, T. K. Leen, and K. R. Müller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12. MIT Press.
- Baudat, G. and F. Anouar (2000). Generalized discriminant analysis using a kernel approach. *Neural Computation* **12**(10), 2385–2404.

Textbooks on Kernel Methods include:

- Schölkopf, B. and A. J. Smola (2002). *Learning with Kernels*. MIT Press.
- Herbrich, R. (2002). *Learning Kernel Classifiers*. MIT Press.
- Shawe-Taylor, J. and N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press (slides)

Dual Representation

Dual Representation

- Many linear models for regression and classification can be reformulated in terms of a dual representation in which the kernel function arises naturally.

- Consider a linear regression model with a regularized cost functional $(\lambda > 0)$.

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_n) - t_n\right)^2 + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w}$$

- Set the gradient of the first term to zero:

$$\boldsymbol{w} = -\frac{1}{\lambda}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_n) - t_n\right)\boldsymbol{\phi}(\boldsymbol{x}_n) = \sum_{n=1}^{N} a_n\boldsymbol{\phi}(\boldsymbol{x}_n) = \boldsymbol{\Phi}^T\boldsymbol{a}$$

$$a_n \equiv -\frac{1}{\lambda}\left(\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_n) - t_n\right), \quad \boldsymbol{a} = \{a_1, \ldots, a_N\}^T$$

- Note that we defined $\boldsymbol{a}$ using only the 1st term in $J(\boldsymbol{w})$.

Dual Representation

$$J(\boldsymbol{w})=\frac{1}{2}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{\phi}(x_n)-t_n\right)^2+\frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w},\ \boldsymbol{w}=-\frac{1}{\lambda}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{\phi}(x_n)-t_n\right)\boldsymbol{\phi}(x_n)=\boldsymbol{\Phi}^T\boldsymbol{a}$$

- Substitute $\boldsymbol{w}$ in the cost functional:

$$J(\boldsymbol{a})=\frac{1}{2}\sum_{n=1}^{N}\left(\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\phi}(x_n)-t_n\right)^2+\frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a}$$

$$=\frac{1}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\sum_{n=1}^{N}\boldsymbol{\phi}(x_n)\boldsymbol{\phi}(x_n)^T\boldsymbol{\Phi}^T\boldsymbol{a}-\boldsymbol{a}^T\boldsymbol{\Phi}\sum_{n=1}^{N}\boldsymbol{\phi}(x_n)t_n+\frac{1}{2}\boldsymbol{t}^T\boldsymbol{t}+\frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a}$$

$$=\frac{1}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a}-\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{t}+\frac{1}{2}\boldsymbol{t}^T\boldsymbol{t}+\frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a}$$

- Here we used from an earlier lecture

$$\boldsymbol{\Phi}=\begin{pmatrix}\boldsymbol{\phi}^T(\boldsymbol{x}_1)\\ \boldsymbol{\phi}^T(\boldsymbol{x}_2)\\ \vdots\\ \boldsymbol{\phi}^T(\boldsymbol{x}_N)\end{pmatrix}=\begin{pmatrix}\phi_0(\boldsymbol{x}_1) & \phi_1(\boldsymbol{x}_1) & .. & \phi_{M-1}(\boldsymbol{x}_1)\\ \phi_0(\boldsymbol{x}_2) & \phi_1(\boldsymbol{x}_2) & .. & \phi_{M-1}(\boldsymbol{x}_2)\\ \vdots & \vdots & \vdots & \vdots\\ \phi_0(\boldsymbol{x}_N) & \phi_1(\boldsymbol{x}_N) & .. & \phi_{M-1}(\boldsymbol{x}_N)\end{pmatrix},$$

$$\boldsymbol{\Phi}^T\boldsymbol{\Phi}=\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\boldsymbol{\phi}(\boldsymbol{x}_n)^T,$$

$$\boldsymbol{\Phi}^T\boldsymbol{t}=\sum_{n=1}^{N}t_n\boldsymbol{\phi}(\boldsymbol{x}_n)$$

Dual Representation: Gramm Matrix

$$J(\boldsymbol{a}) = \frac{1}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a} - \boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{t} + \frac{1}{2}\boldsymbol{t}^T\boldsymbol{t} + \frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{a}$$

- We can now define the Gramm matrix (symmetric) $\boldsymbol{K}_{N\times N}$ using the kernel function $k(\boldsymbol{x}_n, \boldsymbol{x}_m)$ as follows:

$$K_{nm} = K_{mn} = \boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{\phi}(\boldsymbol{x}_m) \equiv k(\boldsymbol{x}_n, \boldsymbol{x}_m) \quad or \quad \boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^T$$

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^T = \begin{pmatrix} \boldsymbol{\phi}^T(\boldsymbol{x}_1) \\ \boldsymbol{\phi}^T(\boldsymbol{x}_2) \\ \vdots \\ \boldsymbol{\phi}^T(\boldsymbol{x}_N) \end{pmatrix} \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2) & .. & \boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix}$$

- The cost function takes the dual representation form:

$$J(\boldsymbol{a}) = \frac{1}{2}\boldsymbol{a}^T\boldsymbol{K}\boldsymbol{K}\boldsymbol{a} - \boldsymbol{a}^T\boldsymbol{K}\boldsymbol{t} + \frac{1}{2}\boldsymbol{t}^T\boldsymbol{t} + \frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{K}\boldsymbol{a}$$

Solution of the Dual Representation

$$J(\boldsymbol{a}) = \frac{1}{2}\boldsymbol{a}^T\boldsymbol{K}\boldsymbol{K}\boldsymbol{a} - \boldsymbol{a}^T\boldsymbol{K}\boldsymbol{t} + \frac{1}{2}\boldsymbol{t}^T\boldsymbol{t} + \frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{K}\boldsymbol{a}$$

- Setting the gradient wrt $\boldsymbol{a}$ equal to zero:

$$\boldsymbol{a} = \left(\boldsymbol{K} + \lambda \boldsymbol{I}_{N\times N}\right)^{-1}\boldsymbol{t}, \quad K_{nm} = K_{mn} = k(\boldsymbol{x}_n, \boldsymbol{x}_m) \quad or \quad \boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^T \ (N \times N \ matrix)$$

- Using $\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{a}$, the prediction for a new input $\boldsymbol{x}$ is

$$y(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{a}^T\boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\Phi}^T\boldsymbol{a} = \boldsymbol{k}(\boldsymbol{x})^T\left(\boldsymbol{K} + \lambda\boldsymbol{I}_{N\times N}\right)^{-1}\boldsymbol{t}$$

where

$$\boldsymbol{k}(\boldsymbol{x})^T = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\Phi}^T = \left(\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x}_1) \quad \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x}_2) \quad .. \quad \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x}_N)\right)$$

$$= \left(k(\boldsymbol{x}_1, \boldsymbol{x}) \quad k(\boldsymbol{x}_2, \boldsymbol{x}) \quad .. \quad k(\boldsymbol{x}_N, \boldsymbol{x})\right)$$

- We now invert an $N \times N$ matrix (to determine $\boldsymbol{a}$) instead of $M \times M$ (to compute $\boldsymbol{w}$). Everything is in terms of kernels (many choices here, kernel trick) but often $N >> M$.

Summary of the Dual Representation

- The dual formulation allows the solution to the problem to be expressed entirely in terms of the kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$.
- The prediction at $\boldsymbol{x}$ is given by a linear combination of the target values from the training set.

$$y(\boldsymbol{x}) = \boldsymbol{k}(\boldsymbol{x})^T \left(\boldsymbol{K} + \lambda \boldsymbol{I}_{N\times N}\right)^{-1} \boldsymbol{t}$$

$$\boldsymbol{k}(\boldsymbol{x})^T = \left(k(\boldsymbol{x}_1, \boldsymbol{x}) \quad k(\boldsymbol{x}_2, \boldsymbol{x}) \quad .. \quad k(\boldsymbol{x}_N, \boldsymbol{x})\right)$$

- Because we now have to invert a larger matrix $N \times N$, the formulation does not seem practical.
- We now have the choice of working with kernels rather than the basis functions $\boldsymbol{\phi}(\boldsymbol{x})$.
  - ✓ The future space can be infinite dimensional.
- The cost of computing $\boldsymbol{a}$ is $\mathcal{O}(N^3)$, whereas the cost of computing $\boldsymbol{w}$ is $\mathcal{O}(M^3)$. However, prediction using the dual variables takes $\mathcal{O}(N)$ time, while prediction with the primal variables only takes $\mathcal{O}(M)$ time.

Dual of the Dual Representation

- It is not difficult to see that the dual of the dual formulation is given by our original least squares representation for $\boldsymbol{w}$.

- Note that $J(\boldsymbol{a})$ can be written in terms of $\boldsymbol{Ka}, \boldsymbol{K} = \boldsymbol{\Phi\Phi}^T$

$$J(\boldsymbol{a}) = \frac{1}{2}\boldsymbol{a}^T\boldsymbol{\Phi\Phi}^T\boldsymbol{\Phi\Phi}^T\boldsymbol{a} - \boldsymbol{a}^T\boldsymbol{\Phi\Phi}^T\boldsymbol{t} + \frac{1}{2}\boldsymbol{t}^T\boldsymbol{t} + \frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{\Phi\Phi}^T\boldsymbol{a} =$$

$$= \frac{1}{2}\left(\boldsymbol{Ka} - \boldsymbol{t}\right)^T\left(\boldsymbol{Ka} - \boldsymbol{t}\right) + \frac{\lambda}{2}\boldsymbol{a}^T\boldsymbol{Ka}$$

- Recall that:

$$\boldsymbol{K} = \boldsymbol{\Phi\Phi}^T = \begin{pmatrix} \boldsymbol{\phi}^T(\boldsymbol{x}_1) \\ \boldsymbol{\phi}^T(\boldsymbol{x}_2) \\ \vdots \\ \boldsymbol{\phi}^T(\boldsymbol{x}_N) \end{pmatrix} \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1) & \boldsymbol{\phi}(\boldsymbol{x}_2) & .. & \boldsymbol{\phi}(\boldsymbol{x}_N) \end{pmatrix}$$

- Since $N >> M$, $\boldsymbol{K}$ is rank deficient – $N - M$ eigenvectors of $\boldsymbol{K}$ have zero eigenvalues. We can decompose $\boldsymbol{a}$ as: $\boldsymbol{a} = \boldsymbol{a}_{\|} + \boldsymbol{a}_{\perp}$, $\boldsymbol{a}_{\perp}^T\boldsymbol{a}_{\|} = 0$ and $\boldsymbol{K}\boldsymbol{a}_{\perp} = \boldsymbol{0}$. Thus $\boldsymbol{a}_{\perp}$ is not determined by $J(\boldsymbol{a})$. One can set $\boldsymbol{a}_{\perp} = 0$, and thus

$$\boldsymbol{a} = \boldsymbol{a}_{\|} \in span\boldsymbol{K}\ (\boldsymbol{K} = \boldsymbol{\Phi\Phi}^T) \Rightarrow \boldsymbol{a} = \boldsymbol{\Phi}\left(\boldsymbol{\Phi}^T\boldsymbol{v}\right) \Rightarrow \boldsymbol{a} = \boldsymbol{\Phi}\underbrace{\mathrm{u}}_{\boldsymbol{u}}$$

Dual of the Dual Representation

- The cost functional becomes:

$$J(\boldsymbol{u}) = \frac{1}{2}\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{u} - \boldsymbol{t}\right)^T\left(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{u} - \boldsymbol{t}\right) + \frac{\lambda}{2}\boldsymbol{u}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{u}$$

- Since $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ has full rank, we can introduce an equivalent parametrization as

$$\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{u}$$

- The cost functional now becomes:

$$J(\boldsymbol{u}) = \frac{1}{2}\left(\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\right)^T\left(\boldsymbol{\Phi}\boldsymbol{w} - \boldsymbol{t}\right) + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w}$$

- This is our original least squares problem. So the proof is complete.

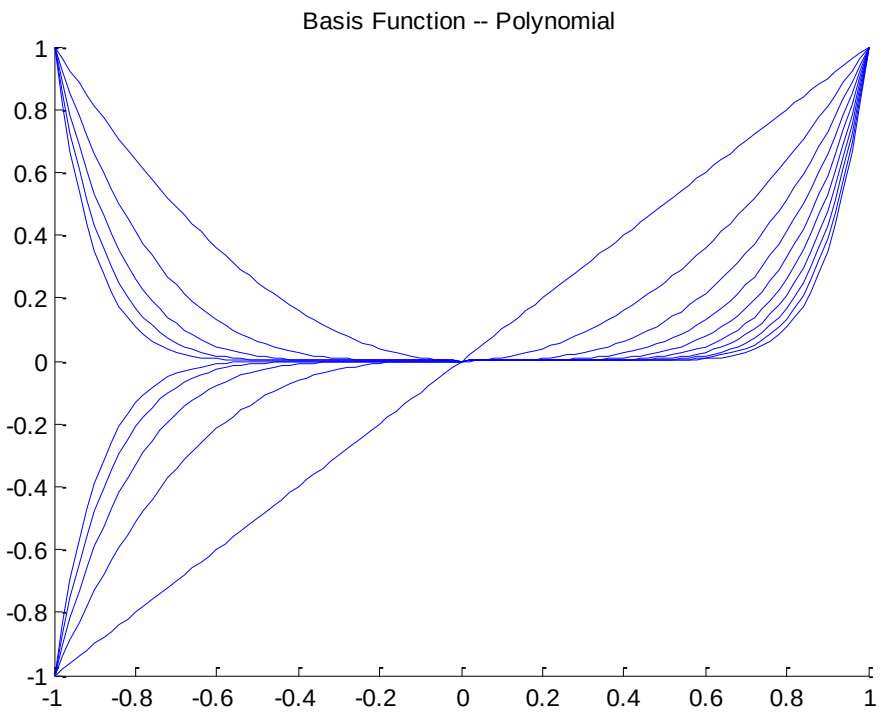
Constructing Kernels from Basis Functions

Constructing Kernels From Basis Functions

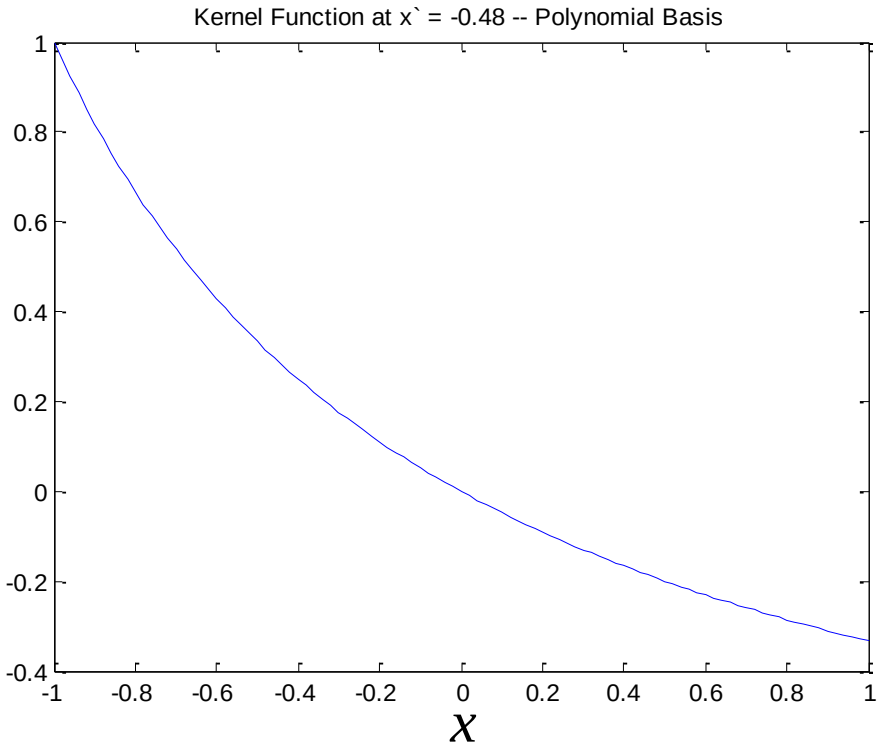
- Choose feature space mapping $\boldsymbol{\phi}(\boldsymbol{x})$, then the kernel (in 1D) is given as:

$$k(x, x') = \boldsymbol{\phi}(x)^T \boldsymbol{\phi}(x') = \sum_{i=1}^{M} \phi_i(x)\phi_i(x')$$

Polynomials



Basis



Kernel $k(x, x')$

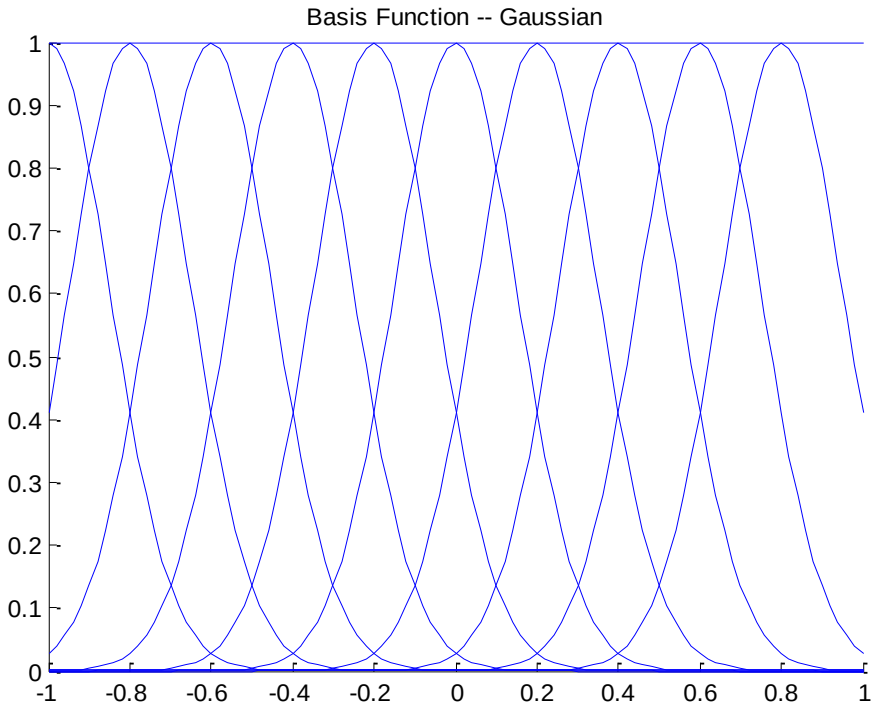
MatLab Code

Constructing Kernels From Basis Functions

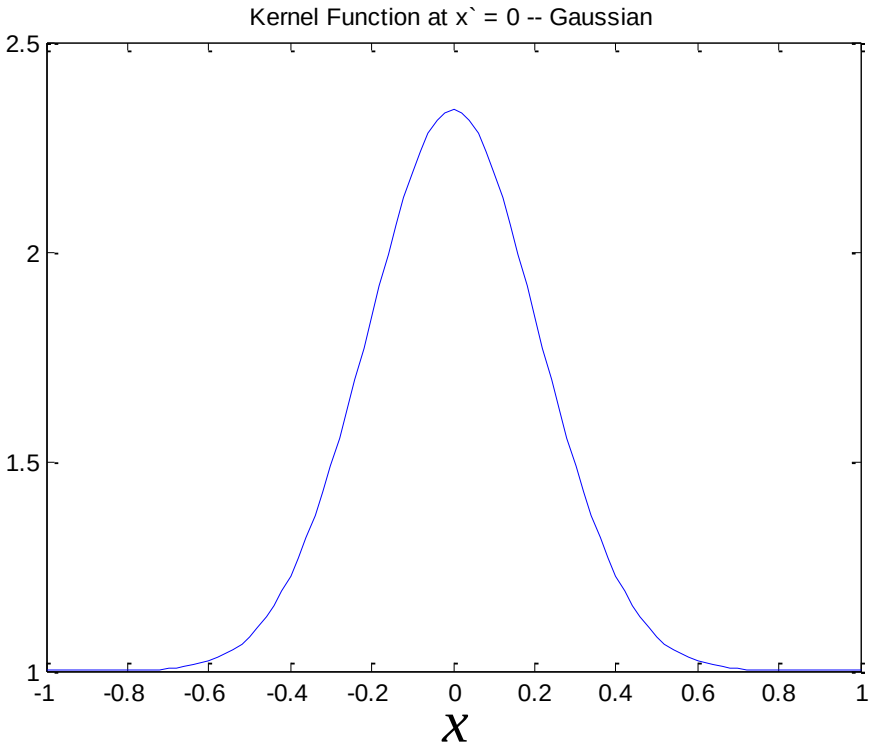
- Choose feature space mapping $\boldsymbol{\phi}(\boldsymbol{x})$, then the kernel (in 1D) is given as:

$$k(x, x') = \boldsymbol{\phi}(x)^T \boldsymbol{\phi}(x') = \sum_{i=1}^{M} \phi_i(x)\phi_i(x')$$

Gaussians



Basis



Kernel $k(x, x')$

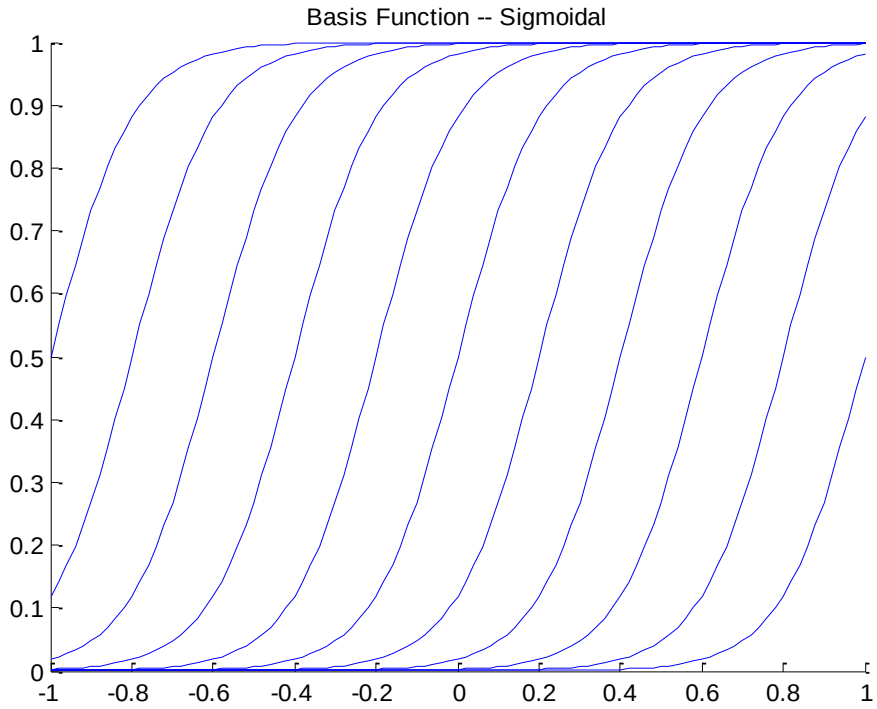
MatLab Code

Constructing Kernels From Basis Functions

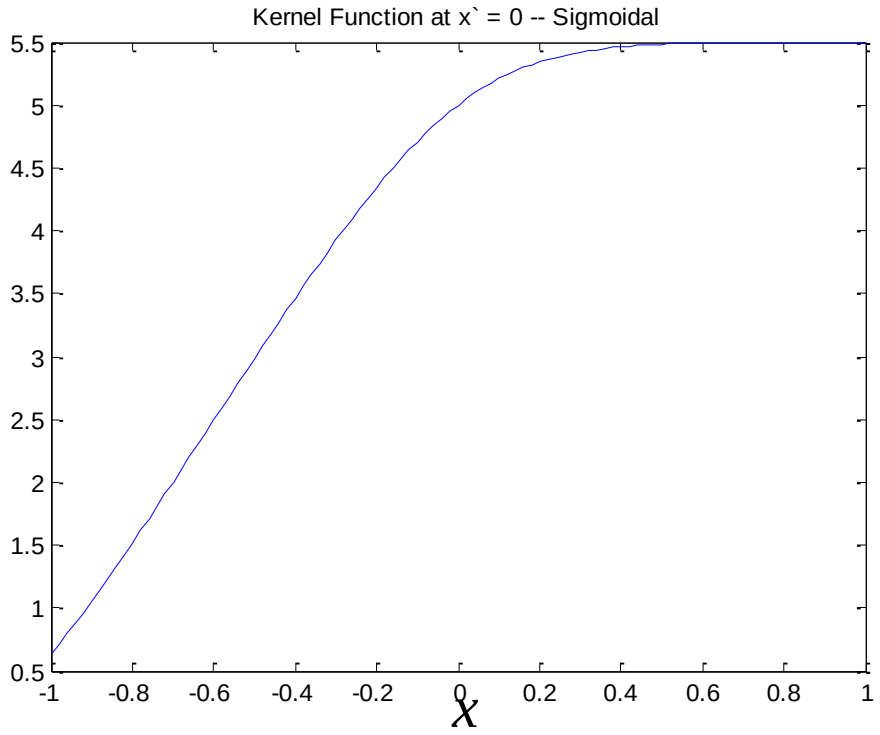
- Choose feature space mapping $\boldsymbol{\phi}(\boldsymbol{x})$, then the kernel (in 1D) is given as:

$$k(x, x') = \boldsymbol{\phi}(x)^T \boldsymbol{\phi}(x') = \sum_{i=1}^{M} \phi_i(x)\phi_i(x')$$

Logistic Sigmoid



Basis



Kernel $k(x, x')$

MatLab Code

Constructing Kernels Directly

- When constructing the kernel functional directly verify its validity
  - **it should correspond to a scalar product in some feature space**
- Simple example

$$k(\boldsymbol{x},\boldsymbol{z}) = \left(\boldsymbol{x}^T\boldsymbol{z}\right)^2$$

- In the 2-D case this corresponds to

$$k(\boldsymbol{x},\boldsymbol{z}) = \left(x_1 z_1 + x_2 z_2\right)^2 = \boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{z})$$

where

$$\boldsymbol{\phi}(\boldsymbol{x}) = \left(x_1^2, \sqrt{2}x_1x_2, x_2^2\right)^T$$

- To **test validity without having to construct $\boldsymbol{\phi}(\boldsymbol{x})$ explicitly, one can check if $K_{nm} = k(\boldsymbol{x}_n, \boldsymbol{x}_m)$ is semidefinite:**

Function $k(\boldsymbol{x_n}, \boldsymbol{x}_m)$ is a valid kernel <-> $K \geq 0$ (Gram matrix) for all $\{x_n\}$

- Shawe-Taylor, J. and N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press (slides)

Kernel Engineering

Kernel Engineering

- Build kernels using simple kernels as building blocks. Given valid kernels $k_1(\boldsymbol{x}, \boldsymbol{x}'), k_2(\boldsymbol{x}, \boldsymbol{x}')$, the following kernels are valid:

$$k(\boldsymbol{x}, \boldsymbol{x}') = ck_1(\boldsymbol{x}, \boldsymbol{x}'), \qquad k(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x})k_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}')$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = q\left(k_1(\boldsymbol{x}, \boldsymbol{x}')\right), \qquad k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(k_1(\boldsymbol{x}, \boldsymbol{x}')\right)$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}'), \qquad k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}')k_2(\boldsymbol{x}, \boldsymbol{x}')$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = k_3(\boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x}')), \qquad k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{A}\boldsymbol{x}'$$

$$k(\boldsymbol{x}, \boldsymbol{x}') = k_a(\boldsymbol{x}_a, \boldsymbol{x}_a') + k_b(\boldsymbol{x}_b, \boldsymbol{x}_b'), \qquad k(\boldsymbol{x}, \boldsymbol{x}') = k_a(\boldsymbol{x}_a, \boldsymbol{x}_a')k_b(\boldsymbol{x}_b, \boldsymbol{x}_b')$$

with corresponding conditions:

$c > 0$, $f(.)$ arbitrary, $q(.)$ polynomial with ≥ 0 coefficients,

$\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^M$, $k_3(.,.)$ valid kernel in $\mathbb{R}^M$, A sym. ≥ 0,

$\boldsymbol{x} = \left(\boldsymbol{x}_a, \boldsymbol{x}_b\right)$, k_a, k_b valid kernels

Combining Kernels: Kernel Engineering

- If $k_1(\boldsymbol{x}, \boldsymbol{x}')$ is a valid kernel, so is $k(\boldsymbol{x}, \boldsymbol{x}') = ck_1(\boldsymbol{x}, \boldsymbol{x}'), c > 0$

$$k(\boldsymbol{x}, \boldsymbol{x}') = ck_1(\boldsymbol{x}, \boldsymbol{x}') = c\boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}') = \boldsymbol{u}(\boldsymbol{x})^T \boldsymbol{u}(\boldsymbol{x}'), \text{ where } \boldsymbol{u}(\boldsymbol{x}) = \sqrt{c}\boldsymbol{\phi}(\boldsymbol{x})$$

- If $k_1(\boldsymbol{x}, \boldsymbol{x}')$ is a valid kernel so is $k(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x})k_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}')$ for $f(.)$ arbitrary.

$$k(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x})k_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}') = f(\boldsymbol{x})\boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')f(\boldsymbol{x}') = \boldsymbol{u}(\boldsymbol{x})^T \boldsymbol{u}(\boldsymbol{x}'), \text{ where}$$

$$\boldsymbol{u}(\boldsymbol{x}) = f(\boldsymbol{x})\boldsymbol{\phi}(\boldsymbol{x})$$

- If $k_1(\boldsymbol{x}, \boldsymbol{x}'), k_2(\boldsymbol{x}, \boldsymbol{x}')$ are valid kernels so is $k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}')$
  Indeed, note that the Gram matrix $\boldsymbol{K}$ with elements $k(x_n, x_m)$ for this kernel is ≥ 0.

$$\forall \boldsymbol{a},\ \boldsymbol{a}^T \boldsymbol{K}\boldsymbol{a} = \boldsymbol{a}^T \boldsymbol{K}_1\boldsymbol{a} + \boldsymbol{a}^T \boldsymbol{K}_2\boldsymbol{a} \geq 0, \text{ since } \boldsymbol{K}_1, \boldsymbol{K}_2 \geq 0$$

- Schoelkopf, B. and A. Smola (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

Kernel Examples

- Polynomial kernels:
  - $k(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}'\right)^2$: $\boldsymbol{\phi}(\boldsymbol{x})$ contains only terms of degree 2
  - $k(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}' + \boldsymbol{c}\right)^2, c > 0$: $\boldsymbol{\phi}(\boldsymbol{x})$ contains constant, linear and order 2 terms
  - $k(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}'\right)^M$: contains all monomials of order M

    Example: For $\boldsymbol{x}$ and $\boldsymbol{x}'$ images, $k(\boldsymbol{x}, \boldsymbol{x}')$ is a weighted sum of all possible products of M pixels of image $\boldsymbol{x}$ with M pixels of image $\boldsymbol{x}'$.
  - $k(\boldsymbol{x}, \boldsymbol{x}') = \left(\boldsymbol{x}^T \boldsymbol{x}' + \boldsymbol{c}\right)^M, c > 0$: contains all terms up to order M

Squared Exponential or Gaussian Kernel

- The **squared exponential kernel** (SE kernel) or **Gaussian kernel** is defined by

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}')^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{x}') \right)$$

- If $\boldsymbol{\Sigma}$ is diagonal, this can be written as

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{D} \frac{1}{\sigma_j^2}(x_j - x_j')^2 \right)$$

- σ_j defines the characteristic length scale of dimension j. If $\sigma_j = \infty$, the j −th dimension is ignored; this is known as the ARD kernel.
- If $\boldsymbol{\Sigma}$ is spherical, we get the isotropic kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2} \right)$
- Here σ^2 is known as the bandwidth and the kernel as the radial basis function or RBF kernel, since it is only a function of $||\boldsymbol{x} - \boldsymbol{x}'||$.

RBF or Gaussian Kernel

- Gaussian kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2} \right)$$

- To see that this is a valid kernel note:

$$\|\boldsymbol{x} - \boldsymbol{x}'\|^2 = \boldsymbol{x}^T \boldsymbol{x} + \boldsymbol{x}'^T \boldsymbol{x}' - 2\boldsymbol{x}^T \boldsymbol{x}'$$

and write it as:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2\sigma^2} \left(\boldsymbol{x}^T \boldsymbol{x} + \boldsymbol{x}'^T \boldsymbol{x}' - 2\boldsymbol{x}^T \boldsymbol{x}' \right) \right)$$

$$= \underbrace{\exp\left(-\frac{\boldsymbol{x}^T \boldsymbol{x}}{2\sigma^2} \right)}_{f(\boldsymbol{x})} \underbrace{\exp\left(\frac{\boldsymbol{x}^T \boldsymbol{x}'}{\sigma^2} \right)}_{k_1(\boldsymbol{x}, \boldsymbol{x}')} \underbrace{\exp\left(-\frac{\boldsymbol{x}'^T \boldsymbol{x}'}{2\sigma^2} \right)}_{f(\boldsymbol{x}')}$$

- Expanding the middle exponential, we can see that the Gaussian kernel can be expressed as the inner product of an infinite dimensional feature vector.

Gaussian Kernel: non-Euclidean Distance

- Gaussian kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2\sigma^2} \left(\boldsymbol{x}^T \boldsymbol{x} + \boldsymbol{x}'^T \boldsymbol{x}' - 2\boldsymbol{x}^T \boldsymbol{x}' \right) \right)$$

- With the kernel trick, we can replace the Euclidean distance with any non-linear kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2\sigma^2} \left(\kappa(\boldsymbol{x}, \boldsymbol{x}) + \kappa(\boldsymbol{x}', \boldsymbol{x}') - 2\kappa(\boldsymbol{x}, \boldsymbol{x}') \right) \right)$$

Kernels for Comparing Documents

- Consider documents $\boldsymbol{x}_i$ and $\boldsymbol{x}_{i'}$ using a bag of words representation, x_{ij} is the number of times word j occurs in document i.
- One can use a cosine similarity (cosine of the angle between $\boldsymbol{x}_i, \boldsymbol{x}_{i'}$):

$$k(\boldsymbol{x}_i, \boldsymbol{x}_{i'}) = \frac{\boldsymbol{x}_i^T \boldsymbol{x}_{i'}}{\|\boldsymbol{x}_i\|_2 \|\boldsymbol{x}_{i'}\|_2}$$

- For orthogonal vectors (no common words) $k(\boldsymbol{x}_i, \boldsymbol{x}_{i'}) = 0$. In general

$$0 \leq k(\boldsymbol{x}_i, \boldsymbol{x}_{i'}) \leq 1$$

- This measure unfortunately accounts for document similarity that may be due to the appearance of non discriminative common words (stop words) in documents such as `the' or `and'.
- In addition, if a discriminative word occurs many times in a document, the similarity is artificially boosted (once a word is used in a document it is very likely to be used again).

Kernels for Comparing Documents

- To improve performance we replace the word count x_{ij} with a new feature vector (TF-IDF) - "term frequency - inverse document frequency". Define the TF as:

$$tf(x_{ij}) \equiv log(1 + x_{ij})$$

- This reduces the impact of words that occur many times within one document. The inverse document frequency is defined as

$$idf(j) \equiv log\left(\frac{N}{1 + \sum_{i=1}^{N} \mathbb{I}(x_{ij} > 0)}\right)$$

- N is the total number of documents, and the denominator counts how many documents contain term j. Finally, we define the future space

$$tf - idf(j) \equiv \boldsymbol{\phi}(\boldsymbol{x}_i) \equiv \left[tf(x_{ij}) \times idf(j)\right]_{j=1}^{V}$$

- The kernel is then taken as: $$k(\boldsymbol{x}_i, \boldsymbol{x}_{i'}) = \frac{\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_{i'})}{\|\phi(\boldsymbol{x}_i)\|_2 \|\phi(\boldsymbol{x}_{i'})\|_2}$$

- Elkan, C. (2005). Deriving TF-IDF as a Fisher kernel. In *Proc. Intl. Symp. on String Processing and Information Retrieval (SPIRE)*, pp. 296–301.
- Manning, C. & H. Schuetze (1999) *Foundations of statistical natural language processing* MIT Press.

Matern Kernel

- This is mostly used in Gaussian Processes. It takes the form:

$$k(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{\ell} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu} r}{\ell} \right)$$

- Here K_{ν} is a modified Bessel function, $\nu > 0, \ell > 0, r = \|\boldsymbol{x} - \boldsymbol{x}'\|$
- As $\nu \to \infty$, the Matern kernel approaches the SE kernel.
- For $\nu = \frac{1}{2}$, $k(r) = \exp(-r/\ell)$

  If $D = 1$, the corresponding GP with this kernel is equivalent to the Ornstein-Uhlenbeck process (Brownian motion, continuous but not differentiable)

String Kernels

- s is a substring of x if we can write $x = usv$ for some strings u, s, v.

- Let $\phi_s(x)$ denote the number of times that substring s appears in string x. We define the kernel between two strings x and x' as

$$\kappa(x, x') = \sum_{s \in \mathcal{A}^*} w_s \phi_s(x) \phi_s(x'), w_s \geq 0$$

  and $\mathcal{A}^*$ is the set of all strings (of any length) from the alphabet $\mathcal{A}$. This is a Mercer kernel, and can be computed in $\mathcal{O}(|x| + |x'|)$ time (for certain settings of the weights $\{w_s\}$) using suffix trees.

- For $w_s = 0$ for $|s| > 1$, we get a bag-of-characters kernel. $\phi_1(x)$ is number of times each character in $\mathcal{A}$ occurs in x.

- If s is to be bordered by white-space, we get a bag-of-words kernel, where $\phi_s(x)$ counts how many times each possible word occurs.

String Kernels

$$\kappa(x, x') = \sum_{s \in \mathcal{A}^*} w_s \phi_s(x) \phi_s(x'), w_s \geq 0$$

- If we consider strings of a fixed length k, we get the k-spectrum kernel.
- Can allow character mismatches, string kernels to compare trees, etc.

- Leslie, C., E. Eskin, A. Cohen, J. Weston, and W. Noble (2003). Mismatch string kernels for discriminative protein classification. *Bioinformatics 1*, 1–10.
- Vishwanathan, S. V. N. and A. Smola (2003). Fast kernels for string and tree matching. In *NIPS*.
- Shawe-Taylor, J. and N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. Cambridge.
- Collins, M. and N. Duffy (2002). Convolution kernels for natural language. In *NIPS*.

Kernels for Symbolic Inputs

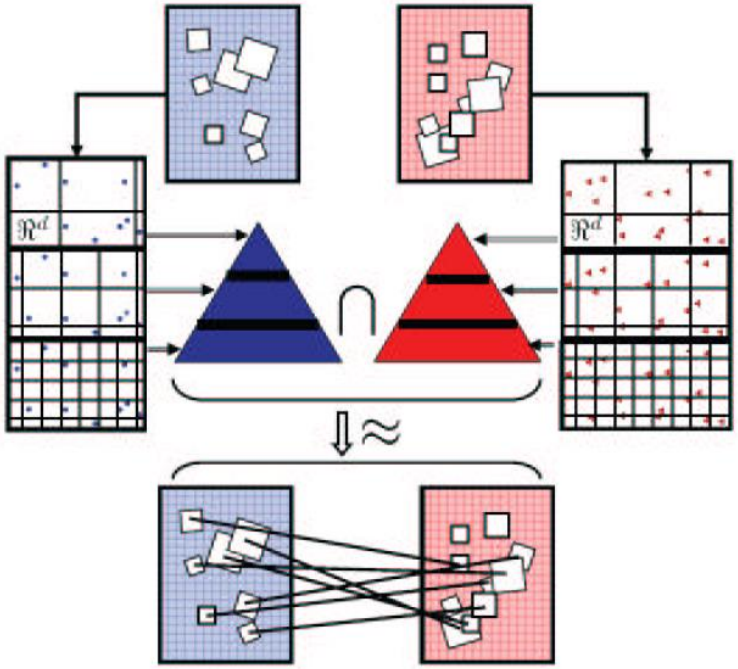
- The inputs can be symbolic, graphs, sets, strings, text, etc.
- For example, fix a set D and consider all of its subsets ($2^{|D|}$). Is A_1 and A_2 are two subsets of D, we can define the kernel as:

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|}$$

  where $|A_1 \cap A_2|$ is the number of elements in the intersection $A_1 \cap A_2$
- $\phi(D)$ is defined to map to a vector of $2^{|D|}$ 1's, one for each possible subset of D, including D itself as well as the empty set.
- For $A \subset D$, $\phi(A)$ will have 1s in all positions that correspond to subsets of A and 0's in all other positions. Therefore, $\phi(A_1)^T \phi(A_2)$ will count the number of subsets shared by A_1 and A_2.
- This can also be obtained by counting the number of elements in the intersection of A_1 and A_2, and then raising 2 to this number, which is exactly $k(A_1, A_2) = 2^{|A_1 \cap A_2|}$

Pyramid Match Kernels

- Each feature set is mapped to a multi-resolution histogram.
- These are then compared using weighted histogram intersection.
- This provides a good approximation to the similarity measure one would obtain by performing an optimal bipartite match at the finest spatial resolution, and then summing up pairwise similarities between matched points.
- The histogram method is faster and is more robust to missing and unequal numbers of points.



- Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Proc. of the International Conference on Computer Vision ICCV, Corfu*, pp. 1150–1157.
- Grauman, K. and T. Darrell (2007, April). The Pyramid Match Kernel: Efficient Learning with Sets of Features. *J. of Machine Learning Research 8*, 725–760.

Probabilistic Generative Models

- Use a generative model to define a kernel and then use this kernel in a discriminative approach.
- Kernel for generative model $p(\boldsymbol{x})$:

$$k(\boldsymbol{x}, \boldsymbol{x}') = p(\boldsymbol{x})\, p(\boldsymbol{x}')$$

- This is a valid kernel function since it is an inner product in the 1D feature space defined by the mapping $p(\boldsymbol{x})$.
- It says that two inputs $\boldsymbol{x}$ and $\boldsymbol{x}'$ are similar if they both have high probabilities.

- Haussler, D. (1999). Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, Computer Science Department.
- Lasserre, J., C. M. Bishop, and T. Minka (2006). Principled hybrids of generative and discriminative models. In *Proceedings 2006 IEEE Conference on Computer Vision and Pattern Recognition, New York*.
- Jebara, T., R. Kondor, and A. Howard (2004). Probability product kernels. *J. of Machine Learning Research 5*, 819–844.

Probabilistic Generative Models

- Using $k(\boldsymbol{x}, \boldsymbol{x}') = p(\boldsymbol{x})p(\boldsymbol{x}')$ and the kernel engineering formulas

$$k(\boldsymbol{x}, \boldsymbol{x}') = ck_1(\boldsymbol{x}, \boldsymbol{x}'), c > 0$$
$$k(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}')$$

we can generalize as:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_i p(\boldsymbol{x} \mid i)\, p(\boldsymbol{x}' \mid i) p(i)$$
$$k(\boldsymbol{x}, \boldsymbol{x}') = \int p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{x}' \mid \boldsymbol{z})\, p(\boldsymbol{z}) d\boldsymbol{z}$$

i and z are latent variables.

- $\boldsymbol{x}$ and $\boldsymbol{x}$' will give large value of $k(\boldsymbol{x}, \boldsymbol{x}')$ if they have significant probability under a range of components.

Probabilistic Generative Models

- Now consider an observation is given by a sequence $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_L\}$.
- A popular generative model for sequences is the Hidden Markov model (HMM), which expresses the distribution $p(\boldsymbol{X})$ as a marginalization over a corresponding sequence of hidden states $\boldsymbol{Z} = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_L\}$.
- We define a kernel function measuring the similarity of two sequences $\boldsymbol{X}$ and $\boldsymbol{X}'$ by extending the earlier mixture representation

$$k(\boldsymbol{X}, \boldsymbol{X}') = \sum_{\boldsymbol{Z}} p(\boldsymbol{X} \mid \boldsymbol{Z}) p(\boldsymbol{X}' \mid \boldsymbol{Z}) p(\boldsymbol{Z})$$

- Note that both observed sequences $\boldsymbol{X}, \boldsymbol{X}'$ are generated by the same hidden sequence $\boldsymbol{Z}$.

Probabilistic Product Kernels

- Jebara et al. defined the so called probability product kernel as follows:

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \int p(\boldsymbol{x} \mid \boldsymbol{x}_i)^{\rho} \, p(\boldsymbol{x} \mid \boldsymbol{x}_j)^{\rho} \, d\boldsymbol{x}, \; \rho > 0$$

- $p(\boldsymbol{x}|\boldsymbol{x}_i)$ is often approximated by $p(\boldsymbol{x}|\widehat{\boldsymbol{\theta}}(\boldsymbol{x}_i))$, where $\widehat{\boldsymbol{\theta}}(\boldsymbol{x}_i)$ is a parameter estimate computed using a single data vector.

- Note that the fitted model to a single data point is only being used to see how similar two objects are. If we fit the model to $\boldsymbol{x}_i$ and then the model thinks $\boldsymbol{x}_j$ is likely, this means that $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are similar.

- For example, let $p(\boldsymbol{x}|\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\mu}, \sigma^2 \boldsymbol{I})$, where σ^2 is constant. If $\rho = 1$, and we use $\widehat{\boldsymbol{\mu}}(\boldsymbol{x}_i) = \boldsymbol{x}_i$ and $\widehat{\boldsymbol{\mu}}(\boldsymbol{x}_j) = \boldsymbol{x}_j$, then we obtain an RBF kernel:

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{1}{\left(4\pi\sigma^2\right)^{D/2}} \exp\left(-\frac{\left\| \boldsymbol{x}_i - \boldsymbol{x}_j \right\|^2}{4\sigma^2} \right)$$

- Jebara, T., R. Kondor, and A. Howard (2004). Probability product kernels. *J. of Machine Learning Research 5*, 819–844.

Mercer (Positive Definite) Kernels

- We define a Mercer (or positive definite) kernel as one for which the Gram matrix $\boldsymbol{K}$ is positive definite for all inputs $\boldsymbol{x}_i, i = 1, \ldots, N$

$$K = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & & \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

- Both the Gaussian and cosine similarity kernels are Mercer kernels.

- Schoelkopf, B. and A. Smola (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.
- Sahami, M. and T. Heilman (2006). A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets. In *WWW conference*.

Mercer (Positive Definite) Kernels

- **Mercer's theorem**. If the Gram matrix is positive definite, we can compute an eigenvector decomposition of it as follows

$$\boldsymbol{K} = \boldsymbol{U}^T \boldsymbol{\Lambda} \boldsymbol{U}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$. We can write:

$$K_{ij} = \left(\boldsymbol{\Lambda}^{\boldsymbol{T}} \boldsymbol{U}_{:i}\right)^T \left(\boldsymbol{\Lambda}^{\boldsymbol{T}} \boldsymbol{U}_{:j}\right) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j), \text{ where } \phi(\boldsymbol{x}_i) \equiv \boldsymbol{\Lambda}^{\boldsymbol{T}} \boldsymbol{U}_{:i}$$

- Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors $\boldsymbol{U}$.

- If the kernel is Mercer, then there exists a function $\boldsymbol{\phi}$ mapping $\boldsymbol{x} \in \mathcal{X}$ to $\mathbb{R}^M$ such that $\mathrm{k}(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}')$, where $\boldsymbol{\phi}$ depends on the eigen *functions* of k (M is a potentially infinite dimensional space).

Mercer Kernels

- Consider the (non-stationary) polynomial kernel $k(\boldsymbol{x}, \boldsymbol{x}') = (\gamma \boldsymbol{x}^T \boldsymbol{x}' + r)^{\mathrm{M}}$ where $r > 0$. The corresponding feature vector $\boldsymbol{\phi}(\boldsymbol{x})$ contains all terms up to degree M. E.g., for $M = 2, \gamma = r = 1$ and $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^2$, we have

$$(1 + \boldsymbol{x}^T \boldsymbol{x}')^2 =$$

$$(1 + x_1 x_1' + x_2 x_2')^2 = 1 + 2x_1 x_1' + 2x_2 x_2' + (x_1 x_1')^2 + (x_2 x_2')^2 + 2x_1 x_1' x_2 x_2'$$

- Thus $\boldsymbol{\phi}(\boldsymbol{x}) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2\right)^T$ i.e. this kernel is equivalent to working in a 6-dimensional feature space.

- In the case of a Gaussian kernel, the feature map lives in an infinite dimensional space.

- An example of a kernel that is not a Mercer kernel ($\boldsymbol{K}$ is not positive semi-definite) is the so-called **sigmoid kernel**, defined by

$$\mathrm{k}(\boldsymbol{x}, \boldsymbol{x}') = \tanh(\gamma \boldsymbol{x}^T \boldsymbol{x}' + r)$$

- Can build up new Mercer kernels from simpler ones using a set of standard rules. E.g., if k_1 and k_2 are both Mercer, so is $\mathrm{k}(\boldsymbol{x}, \boldsymbol{x}') = k_1(\boldsymbol{x}, \boldsymbol{x}') + k_2(\boldsymbol{x}, \boldsymbol{x}')$.

Fisher Kernel

- Consider a parametric generative model $p(\boldsymbol{x}|\boldsymbol{\theta})$. We want a kernel that measures the similarity of two input vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ induced by this distribution.
- The Fisher score is introduced defining a vector in future space

$$g\left(\boldsymbol{\theta}, x\right) = \nabla_{\theta} \ln p\left(\boldsymbol{x} \mid \boldsymbol{\theta}\right)$$

- The Fisher kernel is defined as

$$k(\boldsymbol{x}, \boldsymbol{x}') = g(\boldsymbol{\theta}, \boldsymbol{x})^T \mathbf{F}^{-1} g(\boldsymbol{\theta}, \boldsymbol{x}')$$

The Fisher information matrix $\boldsymbol{F}$ (considers the geometry of the parameter/future space) is given as the expectation under $p\left(\boldsymbol{x} \mid \boldsymbol{\theta}\right)$

$$\mathbf{F} = \mathbb{E}_{\boldsymbol{x}}\left[g(\boldsymbol{\theta}, \boldsymbol{x}) g(\boldsymbol{\theta}, \boldsymbol{x})^T \right]$$

- Using $\boldsymbol{F}$ makes the kernel invariant under $\boldsymbol{\theta} \rightarrow \psi(\boldsymbol{\theta})$ where $\psi(\boldsymbol{\theta})$ is non-linear reparametrization that is invertible & differentiable.

- Jaakkola, T. S. and D. Haussler (1999). Exploiting generative models in discriminative classifiers. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11. MIT Press.
- Amari, S. I. (1998). Natural gradient works efficiently in learning. *Neural Computation* **10**, 251–276.

Fisher Kernel

$$\mathbf{F} = \mathbb{E}_{\boldsymbol{x}}\left[g(\boldsymbol{\theta}, \boldsymbol{x}) g(\boldsymbol{\theta}, \boldsymbol{x})^T \right]$$

- We often approximate the information matrix with a sample average:

$$\boldsymbol{F} \approx \frac{1}{N} \sum_{n=1}^{N} g(\theta, \boldsymbol{x}_n) g(\theta, \boldsymbol{x}_n)^T$$

- This is the covariance matrix of the Fisher scores, and so the Fisher kernel corresponds to a whitening of these scores.

$$k(\boldsymbol{x}, \boldsymbol{x}') = g(\boldsymbol{\theta}, \boldsymbol{x})^T \mathbf{F}^{-1} g(\boldsymbol{\theta}, \boldsymbol{x}')$$

- More simply, we can just omit the Fisher information matrix altogether and use the non-invariant kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = g(\boldsymbol{\theta}, \boldsymbol{x})^T g(\boldsymbol{\theta}, \boldsymbol{x}')$$

- Hofmann, T. (2000). Learning the similarity of documents: an information-geometric approach to document retrieval and classification. In S. A. Solla, T. K. Leen, and K. R. M¨uller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12, pp. 914–920. MIT Press.

Fisher Kernel

- The **score vector** $g(\theta, x)$ as well as the Fisher information matrix F are often computed at the MLE estimate of the parameters $\hat{\theta}$
- Note that $\hat{\theta}$ is a function of all the data, so the similarity of x and x' is computed in the context of all the data as well. Also, note that we only have to fit one model.
- The intuition behind the Fisher kernel is the following: let $g(\hat{\theta}, x)$ be the direction (in parameter space) in which x would like the parameters to move (from $\hat{\theta}$) so as to maximize its own likelihood; call this the directional gradient.
- Then we say that two vectors x and x' are similar if their directional gradients are similar wrt the geometry encoded by the curvature of the likelihood function.

Fisher Kernel

- Interestingly, it was shown that the string kernel discussed earlier is equivalent to the Fisher kernel derived from an L'th order Markov chain.

- Also a kernel defined by the inner product of TF-IDF vectors is approximately equal to the Fisher kernel for a certain generative model of text based on the compound Dirichlet multinomial model.

- Elkan, C. (2005). Deriving TF-IDF as a Fisher kernel. In *Proc. Intl. Symp. on String Processing and Information Retrieval (SPIRE)*, pp. 296–301.
- Saunders, C., J. Shawe-Taylor, and A. Vinokourov (2003). String Kernels, Fisher Kernels and Finite State Automata. In *NIPS*.

Fisher Kernel: Example

- Let us compute the Fisher kernel for the Gaussian distribution with fixed covariance $\boldsymbol{S}$:

$$p(\boldsymbol{x} \mid \boldsymbol{\mu}) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{S})$$

- The Fisher score is given:

$$g\left(\boldsymbol{\mu}, \boldsymbol{x}\right) = \nabla_{\mu} \ln p\left(\boldsymbol{x} \mid \boldsymbol{\mu}\right) = \boldsymbol{S}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})$$

- The Fisher information matrix $\boldsymbol{F}$ is then:

$$\mathbf{F} = \mathbb{E}_{\boldsymbol{x}}\left[g\left(\boldsymbol{\mu}, \boldsymbol{x}\right) g\left(\boldsymbol{\mu}, \boldsymbol{x}\right)^{T} \right] = \boldsymbol{S}^{-1} \underbrace{\mathbb{E}_{\boldsymbol{x}}\left[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^{T} \right]}_{\boldsymbol{S}} \boldsymbol{S}^{-1} = \boldsymbol{S}^{-1}$$

- The Fisher kernel is then given as:

$$k(\boldsymbol{x}, \boldsymbol{x}') = g\left(\boldsymbol{\mu}, \boldsymbol{x}\right)^{T} \mathbf{F}^{-1} g\left(\boldsymbol{\mu}, \boldsymbol{x}'\right) = (\boldsymbol{x} - \boldsymbol{\mu})^{T} \boldsymbol{S}^{-1} (\boldsymbol{x}' - \boldsymbol{\mu})$$

- This is the Mahalanobis distance!

Sigmoidal Kernel

- Another kernel function is the sigmoidal kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \tanh(a\boldsymbol{x}^T \boldsymbol{x}' + b)$$

- Unfortunately the Gram matrix in general is not positive semidefinite.

- This kernel was used because it gives kernel expansions such as the SVM a superficial resemblance to neural network models.

- As we shall see later in this course, in the limit of an infinite number of basis functions, a Bayesian neural network with an appropriate prior reduces to a Gaussian process, thereby providing a link between neural networks and kernel methods.

- Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer

Radial Basis Functions

- By definition

$$\phi_j(\boldsymbol{x}) = h\left(\left\|\boldsymbol{x} - \boldsymbol{\mu}_j\right\|\right)$$

where $\left\|\boldsymbol{x} - \boldsymbol{\mu}_j\right\|$ is a radial distance from a center $\boldsymbol{\mu}_j$.

- Originally were introduced for the problem of exact interpolation $f(\boldsymbol{x}_n) = t_n, n = 1, \ldots, N$

$$f(\boldsymbol{x}) = \sum_{n=1}^{N} w_n h\left(\left\|\boldsymbol{x} - \boldsymbol{x}_n\right\|\right)$$

- There are as many unknowns w_n as target values t_n. Thus least squares leads to exact interpolation.

- If the target values t_n are noisy, exact interpolation overfits.

- Powell, M. J. D. (1987). Radial basis functions for multivariable interpolation: a review. In J. C. Mason and M. G. Cox (Eds.), *Algorithms for Approximation*, pp. 143–167. Oxford University Press.

Radial Basis: Regularized Least Squares

- For a sum-of-squares error function with a regularizer defined in terms of a differential operator, the optimal solution is given by an expansion in the Green's functions of the operator (analogous to eigenvectors of a discrete matrix) again with one basis function centered on each data point.

- Green's functions for an isotropic differential operator as regularizer lead to radial basis functions (depending only on the distance from the data points)

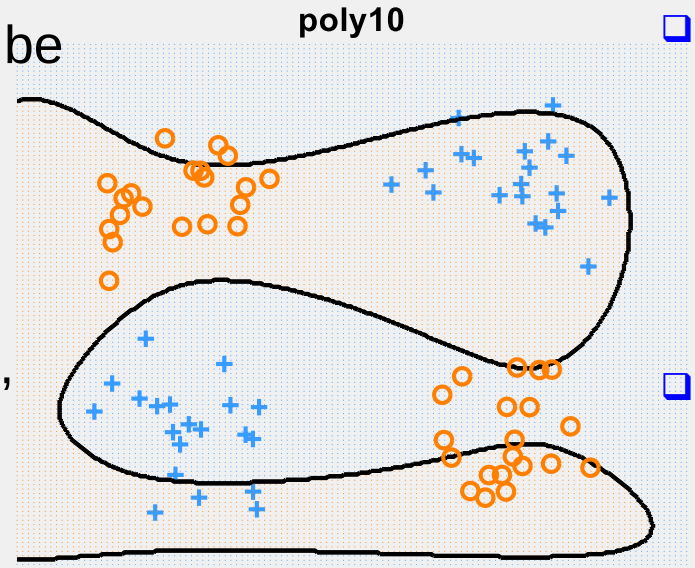
- Interpolation is not exact due to the presence of the regularizer.

- Poggio, T. and F. Girosi (1990). Networks for approximation and learning. *Proceedings of the IEEE* **78**(9), 1481–1497.
- Bishop, C. M. (1995a). *Neural Networks for Pattern Recognition*. Oxford University Press

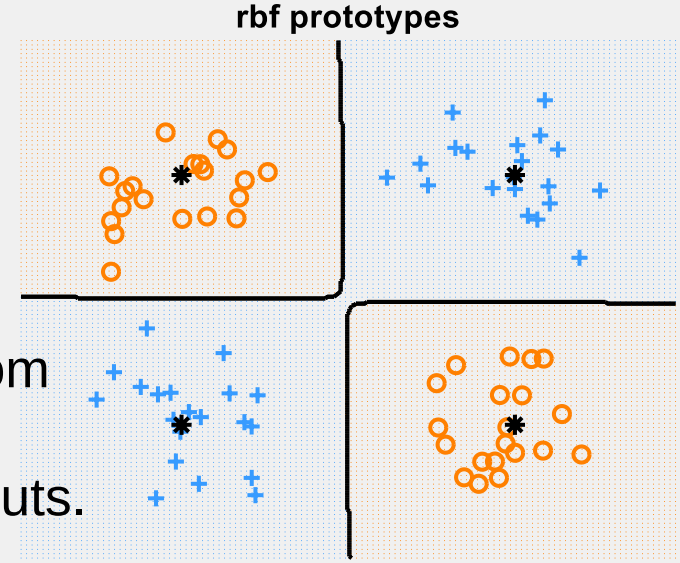
Kernel Machines

Kernel Machines

- We define a **kernel machine** to be a GLM where the input feature vector has the form $\boldsymbol{\phi}(\boldsymbol{x}) = [k(\boldsymbol{x}, \boldsymbol{\mu}_1), \ldots, k(\boldsymbol{x}, \boldsymbol{\mu}_K)]$ where $\boldsymbol{\mu}_i \in X$ are a set of K **centroids**. If k is an RBF kernel, this is called an **RBF network**.
- This defines a kernelized feature vector. The kernel need not be a Mercer kernel.
- Consider the kernelized feature vector for logistic regression by defining $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{B}er(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}))$.
- We consider the data coming from the **xor** function. This is a binary valued function of two binary inputs.



- Fitting a linear logistic regression classifier using degree 10 polynomial expansion.



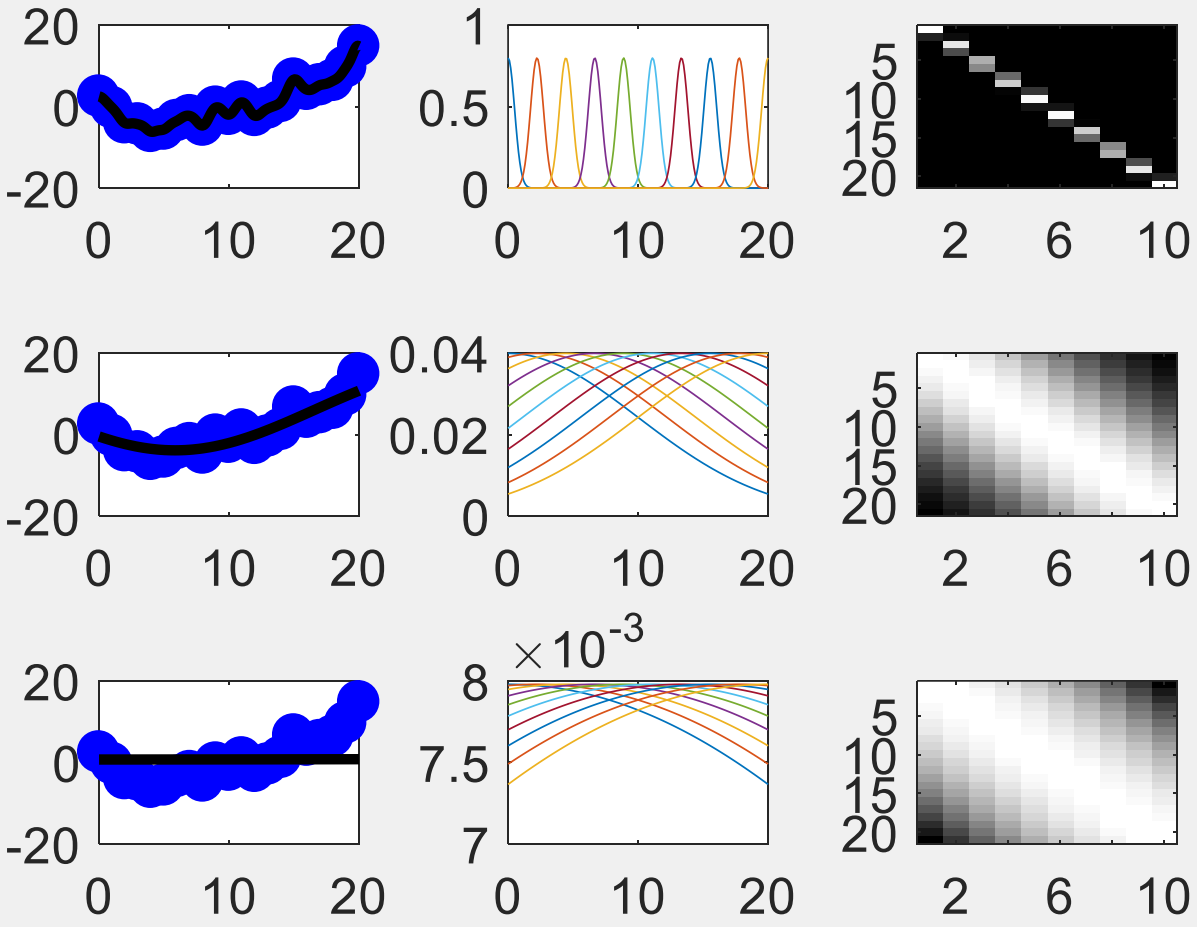
- Same model, but using an RBF kernel with centroids specified by the 4 black crosses.

logregXorDemo.m
from PMTK3

Kernelized Feature Vector in Regression

- RBF basis in 1d.
- Left column: fitted function.
- Middle column: basis functions evaluated on a grid.
- Right column: design matrix.
- K=10 uniformly spaced RBF. Top to bottom we show different bandwidths: $\tau = 0.1, \tau = 0.5, \tau = 50$.

We can use the kernelized feature vector inside a linear regression model by defining $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}), \sigma^2)$

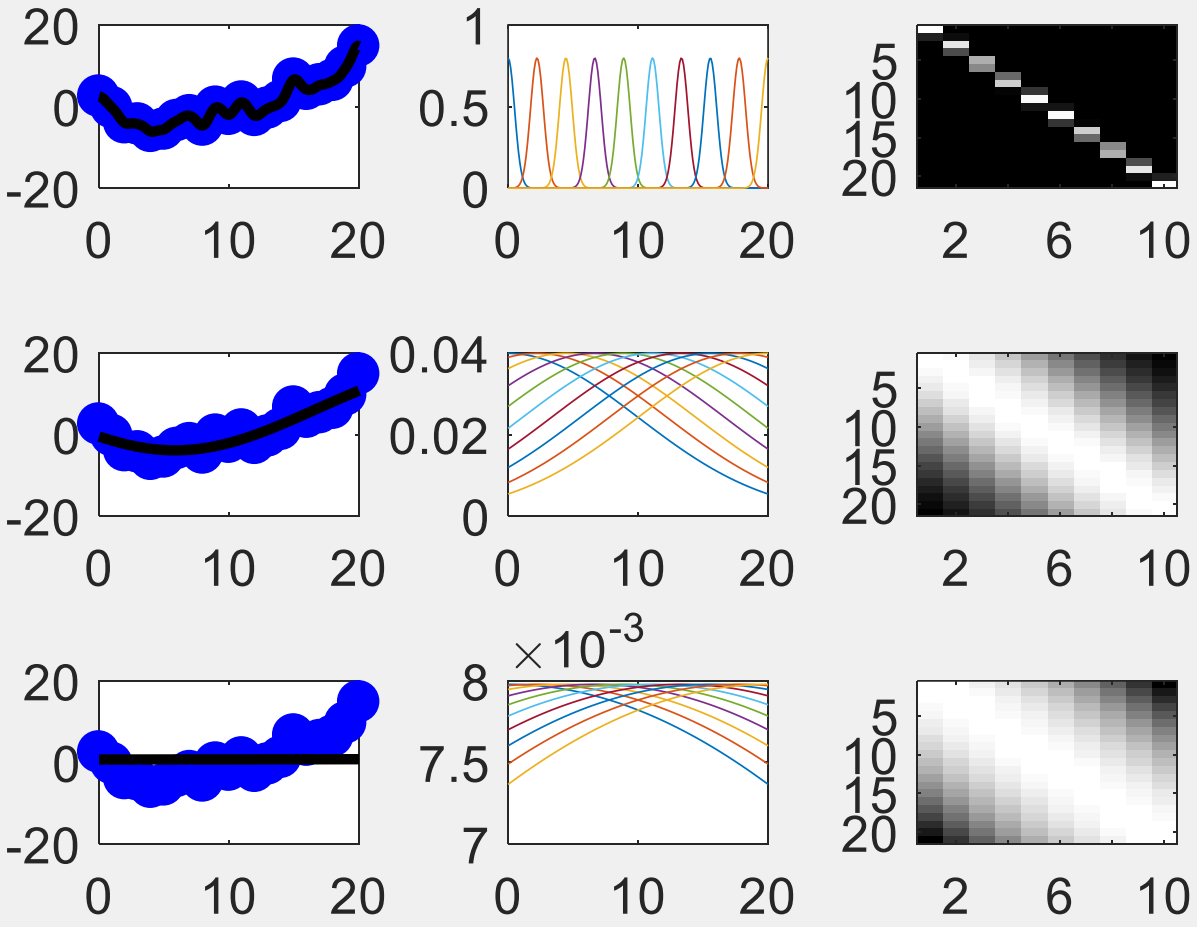


linregRbfDemo.m
from PMTK3

Kernelized Feature Vector in Regression

- Small τ leads to very wiggly functions. The predicted function value will only be non-zero for x that are close to one of the μ_k.
- If τ is very large, the design matrix reduces to a constant matrix of 1's. Each point is equally close to every prototype; the corresponding function is just a straight line.

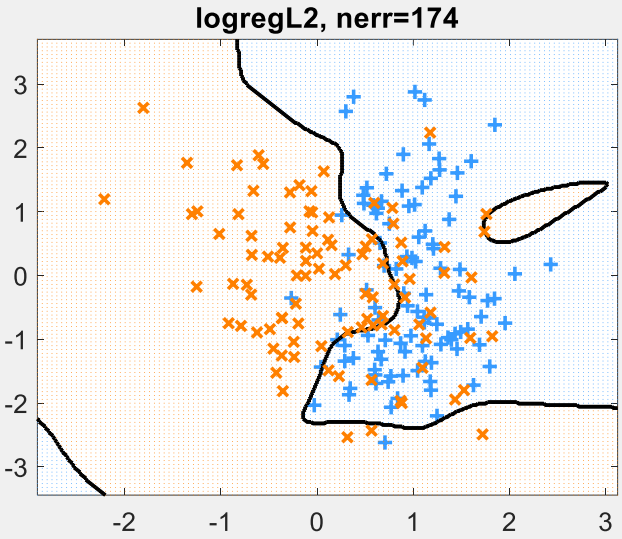
We define $p(y|\boldsymbol{x}, \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}), \sigma^2)$.

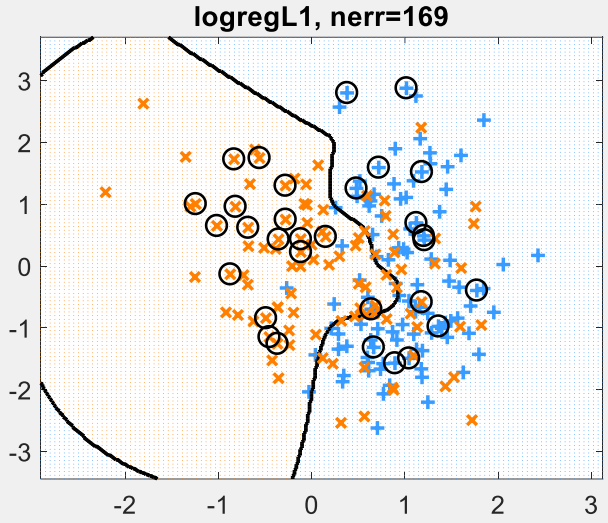


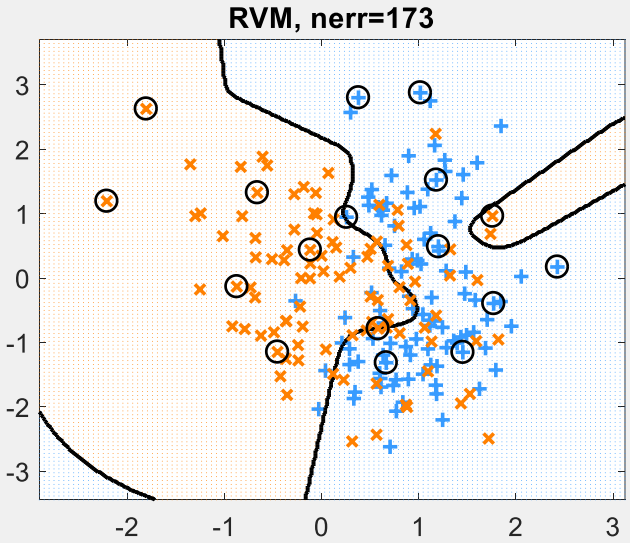
linregRbfDemo.m
from PMTK3

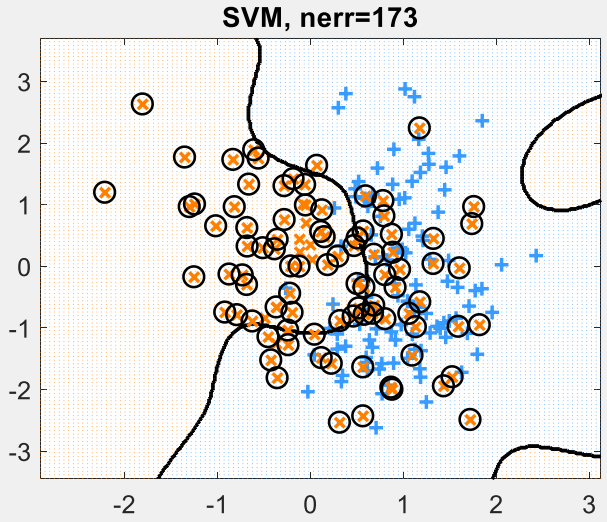
Sparse Vector Machines

- Non-linear binary classification using an RBF kernel with $\sigma = 0.3$.
- (a) L2VM with $\lambda = 5$.
- (b) L1VM with $\lambda = 1$.
- (c) RVM.
- (d) SVM with C = $1/\lambda$ chosen by cross validation.
- Black circles denote the support vectors.
- All the methods give similar performance.
- RVM is the fastest to train and sparsest (and fastest at test time), then L1VM, and then SVM.





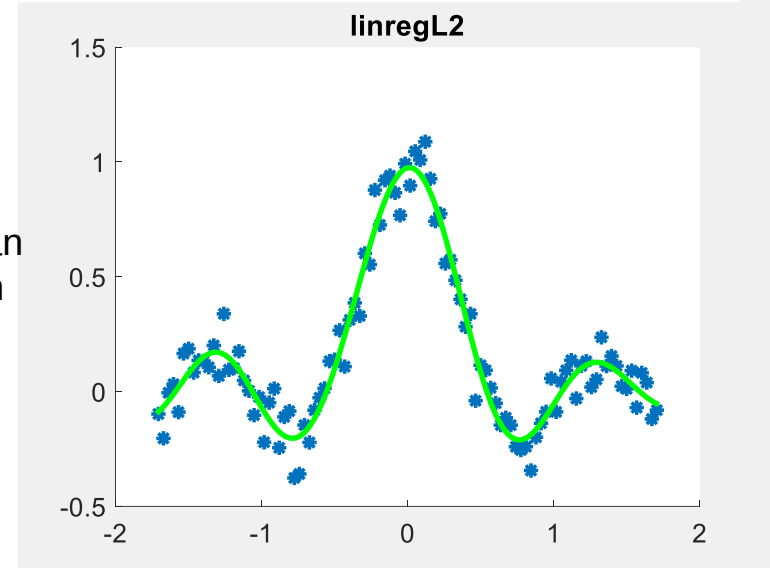


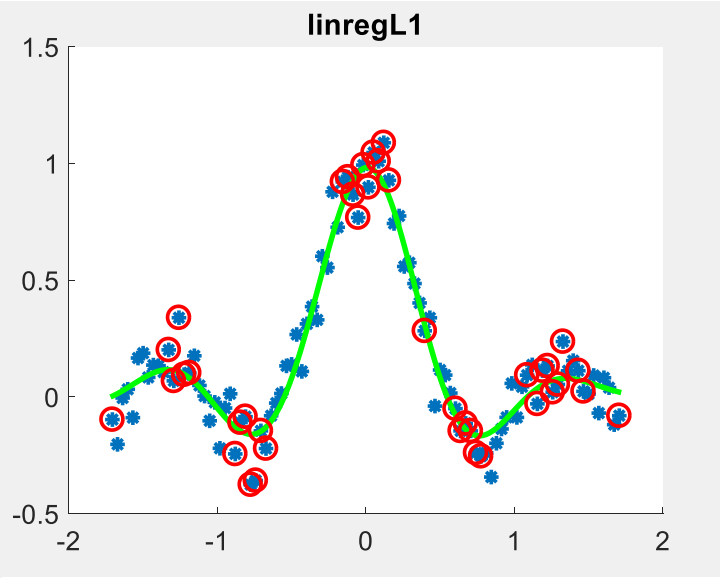


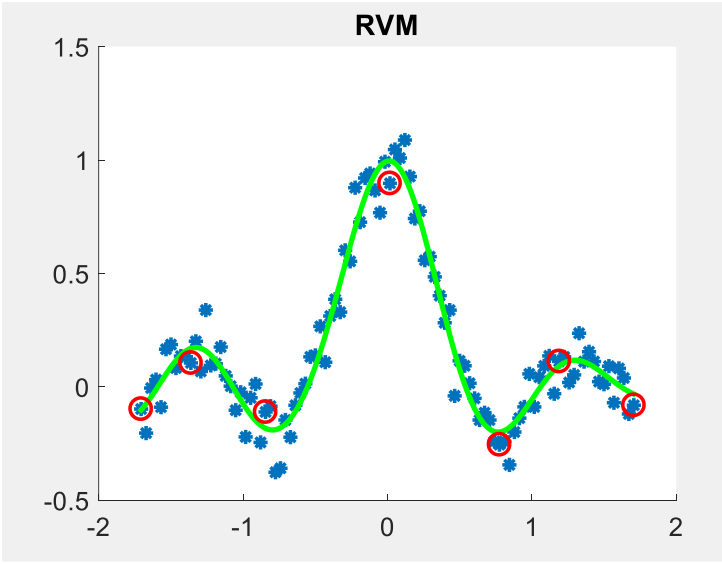
kernelBinaryClassifDemo.m from PMTK3

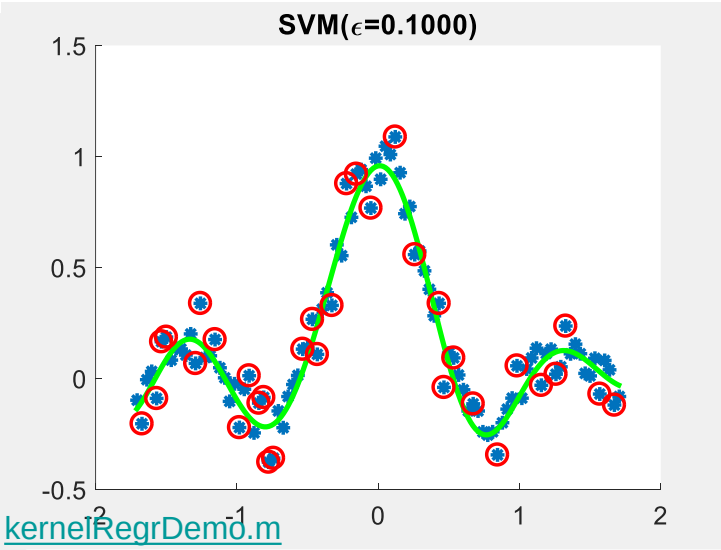
Sparse Vector Machines

- We compare L2VM, L1VM, RVM and an SVM.
- The noisy sinc function using an RBF kernel with $\sigma = 0.3$.
- (a) L2VM with $\lambda = 0.5$.
- (b) L1VM with $\lambda = 0.5$.
- (c) RVM.
- (d) SVM with $C = 1/\lambda$ chosen by cross validation, and $\epsilon = 0.1$. Red circles denote the retained training examples.
- Predictions are similar, but RVM is the sparsest, then L2VM, then SVM.





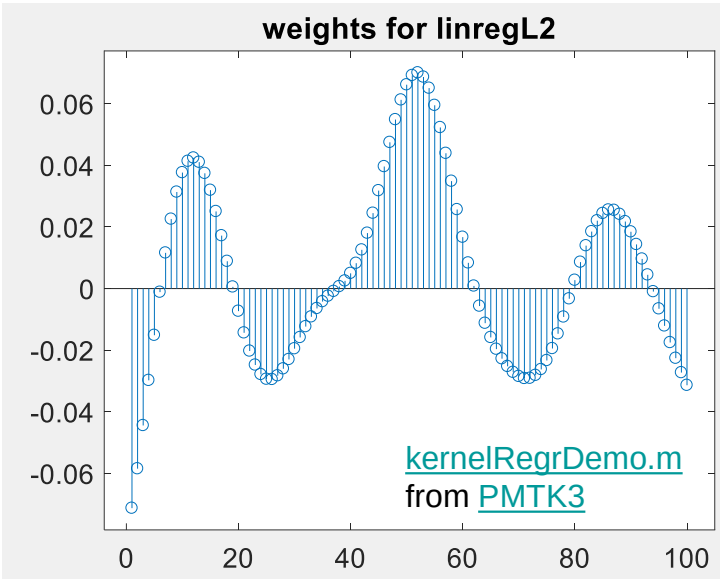


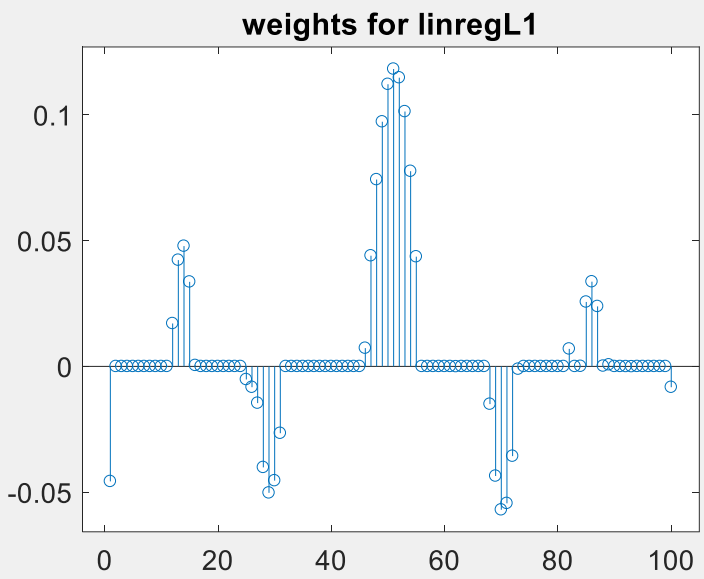


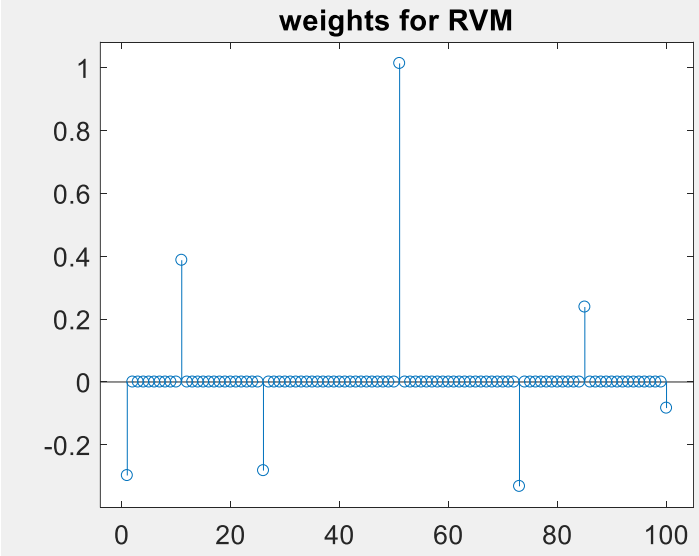
kernelRegrDemo.m from PMTK3

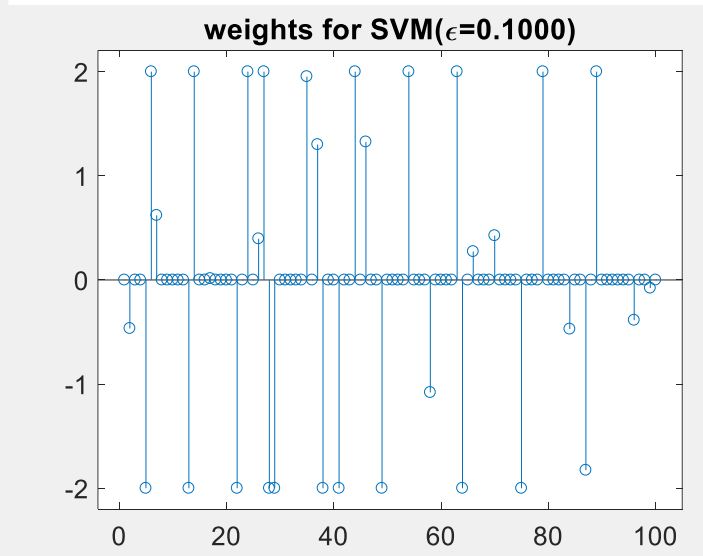
Sparse Vector Machines

- Coefficient vectors of length N=100









Kernel Density Estimation

Smoothing Kernels for Generative Models

- Smoothing kernels are used for non-parametric density estimates. This can be used for unsupervised density estimation, $p(\boldsymbol{x})$, as well as for generative models $p(y, \boldsymbol{x})$.
- A smoothing kernel has one argument and satisfies

$$\int k(x)dx = 1, \int xk(x)dx = 0, \int x^2 k(x)dx > 0$$

- Gaussian kernel:

$$k(x) \equiv \frac{1}{(2\pi)^{1/2}} e^{-x^2/2}$$

$$k_h(x) \equiv \frac{1}{h} k\left(\frac{x}{h}\right), k_h(\boldsymbol{x}) \equiv k_h(\|\boldsymbol{x}\|)$$

$$k_h(\boldsymbol{x}) \equiv \frac{1}{h^D (2\pi)^{D/2}} \prod_{j=1}^{D} \exp\left(-\frac{1}{2h^2} x_j^2\right)$$

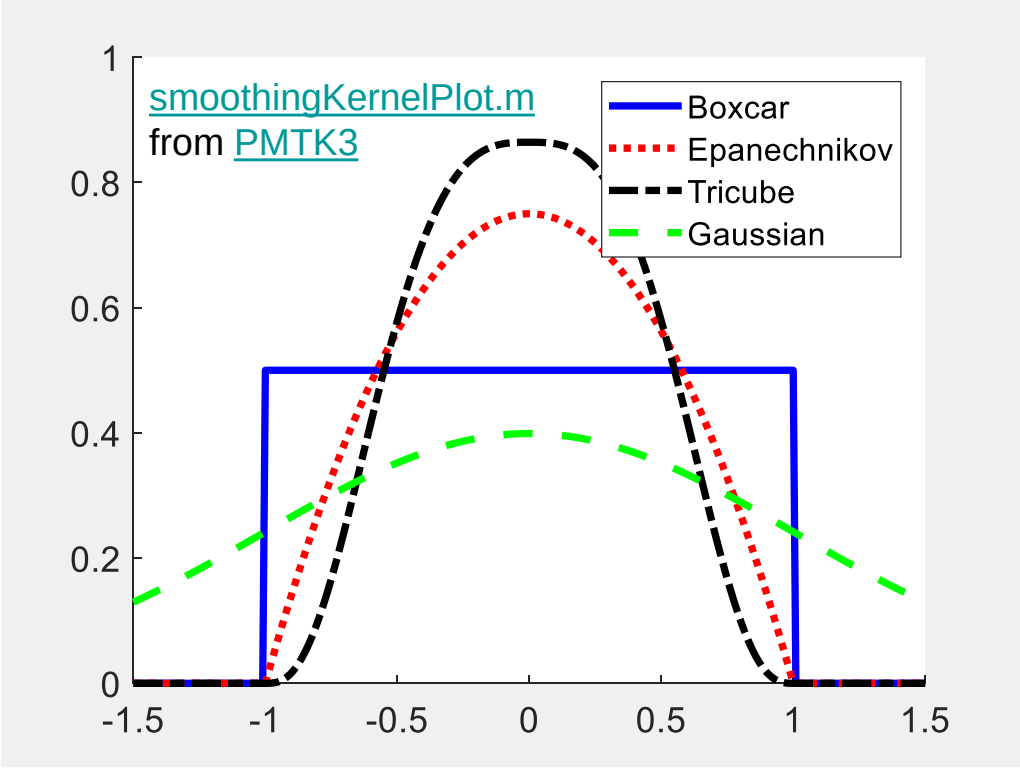
- Epanechnikov kernel:

$$k(x) \equiv \frac{3}{4}(1 - x^2)\mathbb{I}(|x| \leq 1)$$

- Tri-Cube kernel:

$$k(x) \equiv \frac{70}{81}(1 - |x|^3)\mathbb{I}(|x| \leq 1)$$

- Boxcar Kernel: $k(x) \equiv \mathbb{I}(|x| \leq 1)$



Kernel Density Estimation

- One can build a non-parametric density estimator by using a Gaussian mixture with as many components as the available data points, and with means taking to correspond to each data point:

$$p(\boldsymbol{x}|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(\boldsymbol{x}|\boldsymbol{x}_i, \sigma^2\boldsymbol{I})$$

- This can be generalized using smooth kernels as:

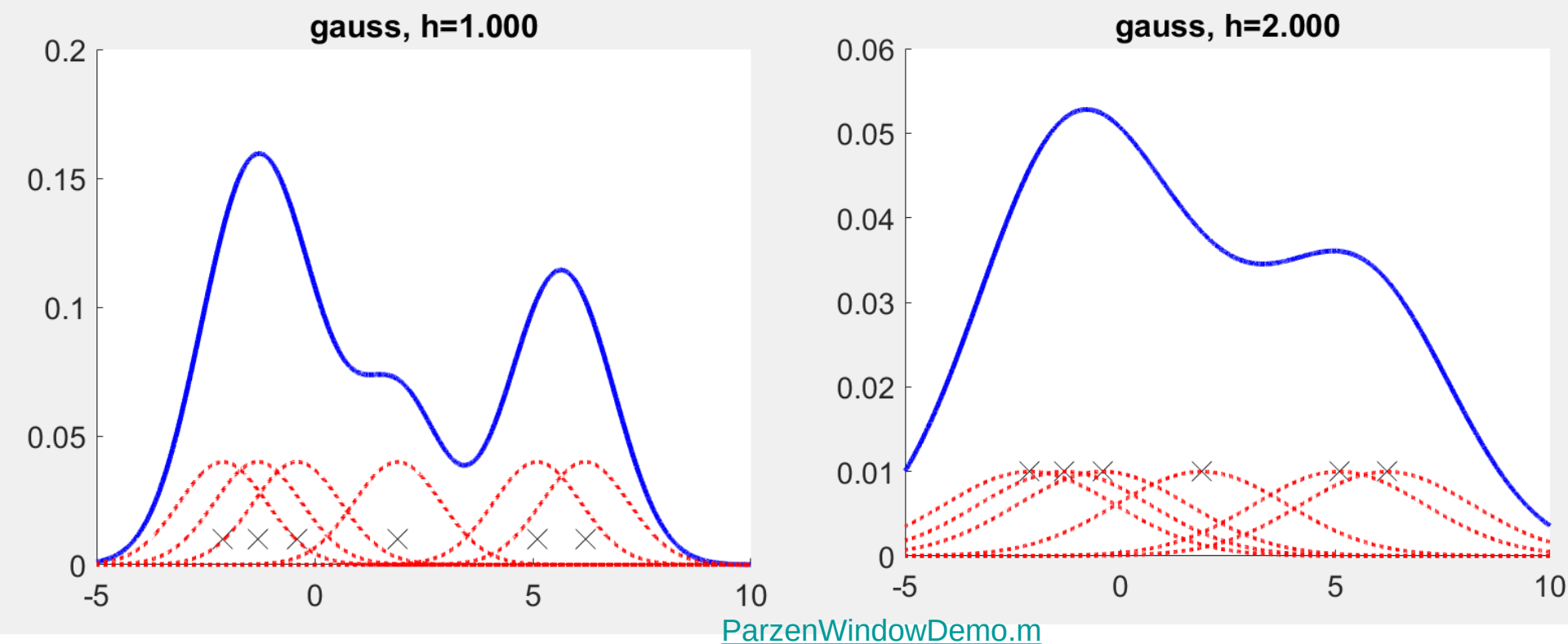
$$p(\boldsymbol{x}|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}k_h(\boldsymbol{x}-\boldsymbol{x}_i)$$

- This is called a Parzen window density estimator, or kernel density estimator (KDE).

- Advantages: No model fitting is required (except for tuning the bandwidth, done by cross-validation) and no need to pick K.

- Disadvantages: Takes a lot of memory to store, and a lot of time to evaluate. It is also of no use for clustering tasks.

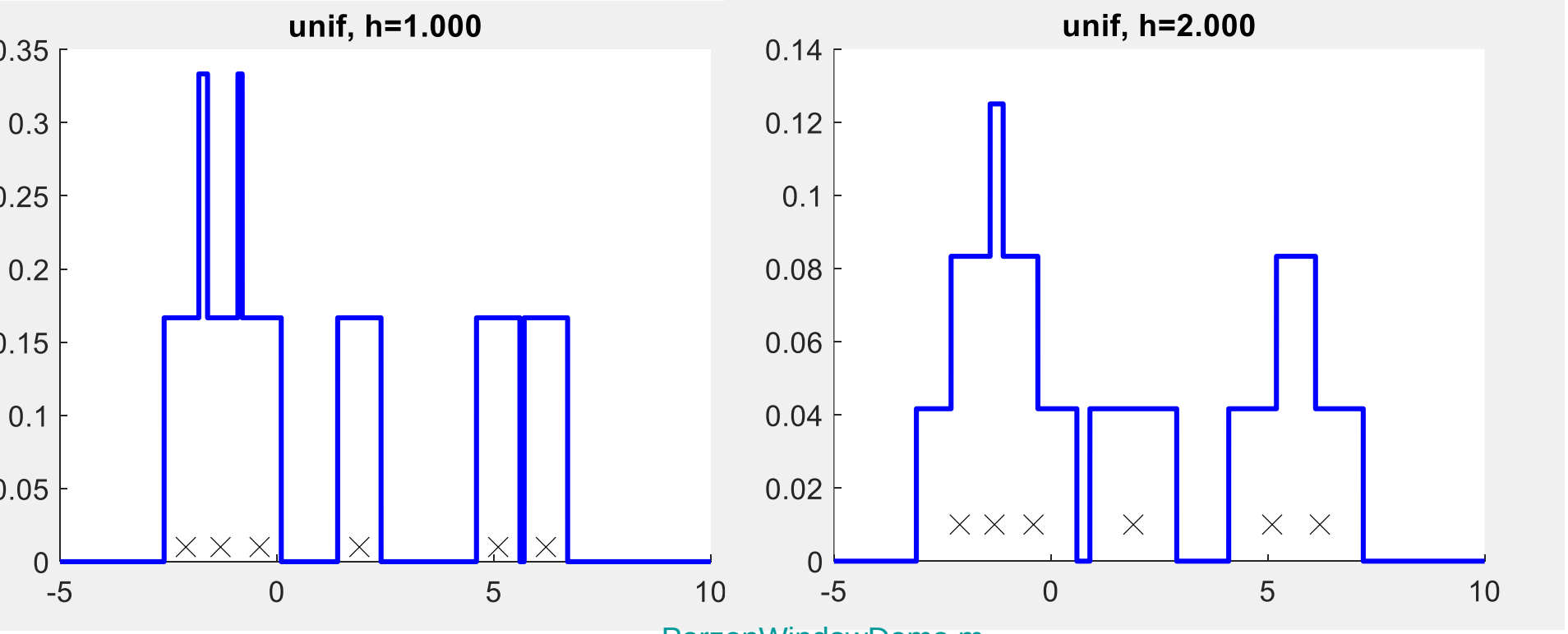
Kernel Density Estimation



ParzenWindowDemo.m
from PMTK3

Kernel Density Estimation

- A boxcar kernel results in a histogram estimation.
- Selection of h can be done with cross validation.



ParzenWindowDemo.m
from PMTK3

- Bowman, A. and A. Azzalini (1997). *Applied Smoothing Techniques for Data Analysis*. Oxford.

KDE for KNN Classifiers

- We use KDE for an alternative derivation of the KNN classifier.
- In KDE with a boxcar kernel, we fix the bandwidth and count how many data points fall within the hyper-cube centered on a datapoint.
- Suppose that we grow a volume around $\boldsymbol{x}$ until we encounter K data points, regardless of their class label. Let the resulting volume have size $V(\boldsymbol{x})$, and let there be $N_c(\boldsymbol{x})$ examples from class c in this volume. We can estimate the class conditional density as follows:

$$p(\boldsymbol{x}|y=c,\mathcal{D}) = \frac{N_c(\boldsymbol{x})}{N_c V(\boldsymbol{x})}$$

  where N_c is the total number of examples in class c in the whole data set. The class prior can be estimated by $p(y=c|\mathcal{D}) = \frac{N_c}{N}$

- Hence the class posterior is given by

$$p(y=c|\boldsymbol{x},\mathcal{D}) = \frac{\frac{N_c(\boldsymbol{x})}{N_c V(\boldsymbol{x})}\frac{N_c}{N}}{\sum_{c'}\frac{N_{c'}(\boldsymbol{x})}{N_{c'} V(\boldsymbol{x})}\frac{N_{c'}}{N}} = \frac{N_c(\boldsymbol{x})}{\sum_{c'} N_{c'}(\boldsymbol{x})} = \frac{N_c(\boldsymbol{x})}{K}$$

Nadaraya-Watson Model

Interpolation with Noisy Inputs

- Consider the interpolation problem with noisy inputs rather than target values:

$$E = \frac{1}{2}\sum_{n=1}^{N}\int \left\{ y\left(\boldsymbol{x}_n + \boldsymbol{\xi}\right) - t_n \right\}^2 \nu(\boldsymbol{\xi}) d\boldsymbol{\xi}$$

- We can optimize with respect to the function $y\left(\boldsymbol{x}\right)$ to obtain:

$$y\left(\boldsymbol{x}\right) = \sum_{n=1}^{N} t_n h\left(\boldsymbol{x} - \boldsymbol{x}_n\right)$$

- The basis functions are given as (Nadaraya-Watson model):

$$h\left(\boldsymbol{x} - \boldsymbol{x}_n\right) = \frac{\nu\left(\boldsymbol{x} - \boldsymbol{x}_n\right)}{\sum_{n=1}^{N} \nu\left(\boldsymbol{x} - \boldsymbol{x}_n\right)}$$

- If noise distribution $\nu(\boldsymbol{\xi})$ is isotropic (function of $\|\boldsymbol{\xi}\|$), these basis functions would be radial.

- Bishop, C. M. (1995a). *Neural Networks for Pattern Recognition*. Oxford University Press
- Webb, A. R. (1994). Functional approximation by feed-forward networks: a least-squares approach to generalisation. *IEEE Transactions on Neural Networks* **5**(3), 363–371.

Radial Basis Functions: Proof

- We take a variation of $y(\boldsymbol{x})$: $y(\boldsymbol{x}) \rightarrow y(\boldsymbol{x}) + \varepsilon\eta(\boldsymbol{x})$. Then:

$$E[y+\varepsilon\eta] = \frac{1}{2}\sum_{n=1}^{N}\int\left\{y(\boldsymbol{x}_n+\boldsymbol{\xi})+\varepsilon\eta(\boldsymbol{x}_n+\boldsymbol{\xi})-t_n\right\}^2\nu(\boldsymbol{\xi})d\boldsymbol{\xi}$$

- We expand in ε and set the coefficient in ε (functional 1st derivative) to zero:

$$\sum_{n=1}^{N}\int\left\{y(\boldsymbol{x}_n+\boldsymbol{\xi})-t_n\right\}\eta(\boldsymbol{x}_n+\boldsymbol{\xi})\nu(\boldsymbol{\xi})d\boldsymbol{\xi} = \boldsymbol{0}$$

- Since this must be true for any variation $\eta(\boldsymbol{x})$, we choose: $\eta(\boldsymbol{x}) = \delta(\boldsymbol{x}-\boldsymbol{z})$. This leads to:

$$\sum_{n=1}^{N}\int\left\{y(\boldsymbol{x}_n+\boldsymbol{\xi})-t_n\right\}\delta(\boldsymbol{x}_n+\boldsymbol{\xi}-\boldsymbol{z})\nu(\boldsymbol{\xi})d\boldsymbol{\xi} = \sum_{n=1}^{N}\left\{y(\boldsymbol{z})-t_n\right\}\nu(\boldsymbol{z}-\boldsymbol{x}_n) = \boldsymbol{0} \Rightarrow$$

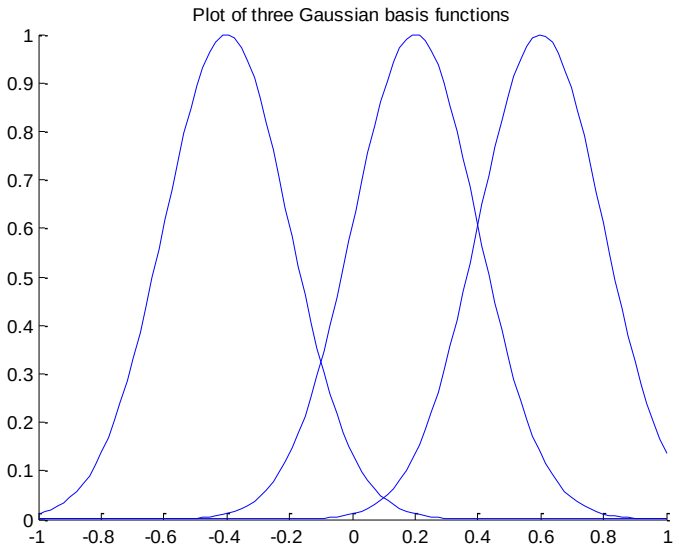
$$y(\boldsymbol{z}) = \sum_{n=1}^{N} t_n \left(\nu(\boldsymbol{z}-\boldsymbol{x}_n)\Big/\sum_{l=1}^{N}\nu(\boldsymbol{z}-\boldsymbol{x}_l)\right)$$

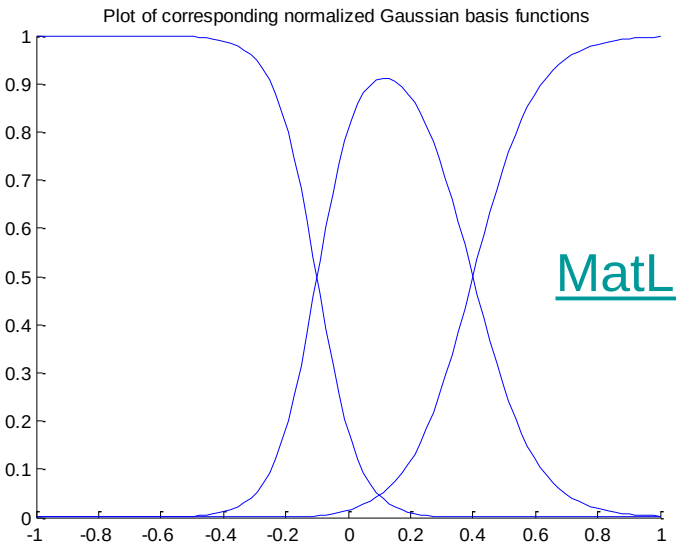
Normalization Effect

- The functions in the Nadaraya-Watson model are normalized:

$$h(\boldsymbol{x} - \boldsymbol{x}_n) = \frac{\nu(\boldsymbol{x} - \boldsymbol{x}_n)}{\sum_{n=1}^{N} \nu(\boldsymbol{x} - \boldsymbol{x}_n)}, \qquad \sum_n h(\boldsymbol{x} - \boldsymbol{x}_n) = 1$$

- It is useful to have normalized basis functions to avoid regions in an input space where all of the basis functions take small values (leading to small predictions or predictions controlled solely by the bias parameter).





MatLab Code

Radial Basis For Regression

- In the application of kernel density estimation to regression, there is one basis function associated with every data point, and the corresponding model can be computationally costly to evaluate when making predictions for new data points.
- Models have therefore been proposed using the expansion in radial basis functions but where the number M of basis functions is smaller than the number N of data points.
- Typically, the number of basis functions, and the centers $\boldsymbol{\mu}_i$ are determined based on the input data $\{\boldsymbol{x}_n\}$ alone. The basis functions are kept fixed and the coefficients $\{w_i\}$ are determined by least squares.

- Broomhead, D. S. and D. Lowe (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems* **2**, 321–355.
- Moody, J. and C. J. Darken (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation* **1**(2), 281–294.
- Poggio, T. and F. Girosi (1990). Networks for approximation and learning. *Proceedings of the IEEE* **78**(9), 1481–1497.

Reducing the Size of the Basis

- To reduce the algorithmic cost, we can keep the number of basis functions M smaller than input data size N
- Centers locations μ_i are determined based on the input data $\{x_n\}$ alone
- Coefficients $\{w_i\}$ are determined by least squares
- For the choice of centers:
  - Random subset of the training data points
  - Orthogonal least squares (sequential) – the next data point to be chosen as a basis function center gives the greatest error reduction
  - Using clustering algorithms (K-means) – basis functions centers no longer coincide with the training data points.

- Chen, S., C. F. N. Cowan, and P. M. Grant (1991). Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks* **2**(2), 302–309.

Parzen Density Estimator

- We have seen in an earlier lecture the prediction of the linear regression model - linear combination of t_n with 'equivalent kernel' values

- The same result is obtained from the Parzen density estimator, $f(\boldsymbol{x}, t)$ is the component density function centered at each training data point

$$p(\boldsymbol{x}, t) = \frac{1}{N} \sum_{n=1}^{N} f(\boldsymbol{x} - \boldsymbol{x}_n, t - t_n)$$

- Regression function

$$y(\boldsymbol{x}) = \mathbb{E}[t \mid \boldsymbol{x}] = \int_{-\infty}^{+\infty} t p(t \mid \boldsymbol{x}) dt =$$

$$= \frac{\int t p(\boldsymbol{x}, t) dt}{\int p(\boldsymbol{x}, t) dt} = \frac{\sum_n \int t f(\boldsymbol{x} - \boldsymbol{x}_n, t - t_n) dt}{\sum_m \int f(\boldsymbol{x} - \boldsymbol{x}_m, t - t_m) dt}$$

Kernel Regression: Nadaraya-Watson Model

- Assume that the component density functions have zero mean so that

$$\int_{-\infty}^{+\infty} t f(\boldsymbol{x},t)dt = 0$$

for all values of $\boldsymbol{x}$. Then by variable change

$$y(\boldsymbol{x}) = \frac{\sum_n \int (t - t_n + t_n) f(\boldsymbol{x} - \boldsymbol{x}_n, t - t_n)dt}{\sum_m \int f(\boldsymbol{x} - \boldsymbol{x}_m, t - t_m)dt} = \frac{\sum_n g(\boldsymbol{x} - \boldsymbol{x}_n)t_n}{\sum_m g(\boldsymbol{x} - \boldsymbol{x}_m)} =$$

$$= \sum_n k(\boldsymbol{x}, \boldsymbol{x}_n)t_n$$

with $g(\boldsymbol{x}) = \int_{-\infty}^{+\infty} f(\boldsymbol{x},t)dt$ and $k(\boldsymbol{x}, \boldsymbol{x}_n)$ given by

$$k(\boldsymbol{x}, \boldsymbol{x}_n) = \frac{g(\boldsymbol{x} - \boldsymbol{x}_n)}{\sum_m g(\boldsymbol{x} - \boldsymbol{x}_m)}$$

Kernel Regression: Nadaraya-Watson Model

- For localized kernels, this regression model gives more weight to the data points $\boldsymbol{x}_n$ close to $\boldsymbol{x}$.

$$y(\boldsymbol{x}) = \sum_n k(\boldsymbol{x}, \boldsymbol{x}_n) t_n$$

$$k(\boldsymbol{x}, \boldsymbol{x}_n) = \frac{g(\boldsymbol{x} - \boldsymbol{x}_n)}{\sum_m g(\boldsymbol{x} - \boldsymbol{x}_m)}$$

- The kernel satisfies the summation constraint:

$$\sum_{n=1}^{N} k(\boldsymbol{x}, \boldsymbol{x}_n) = 1$$

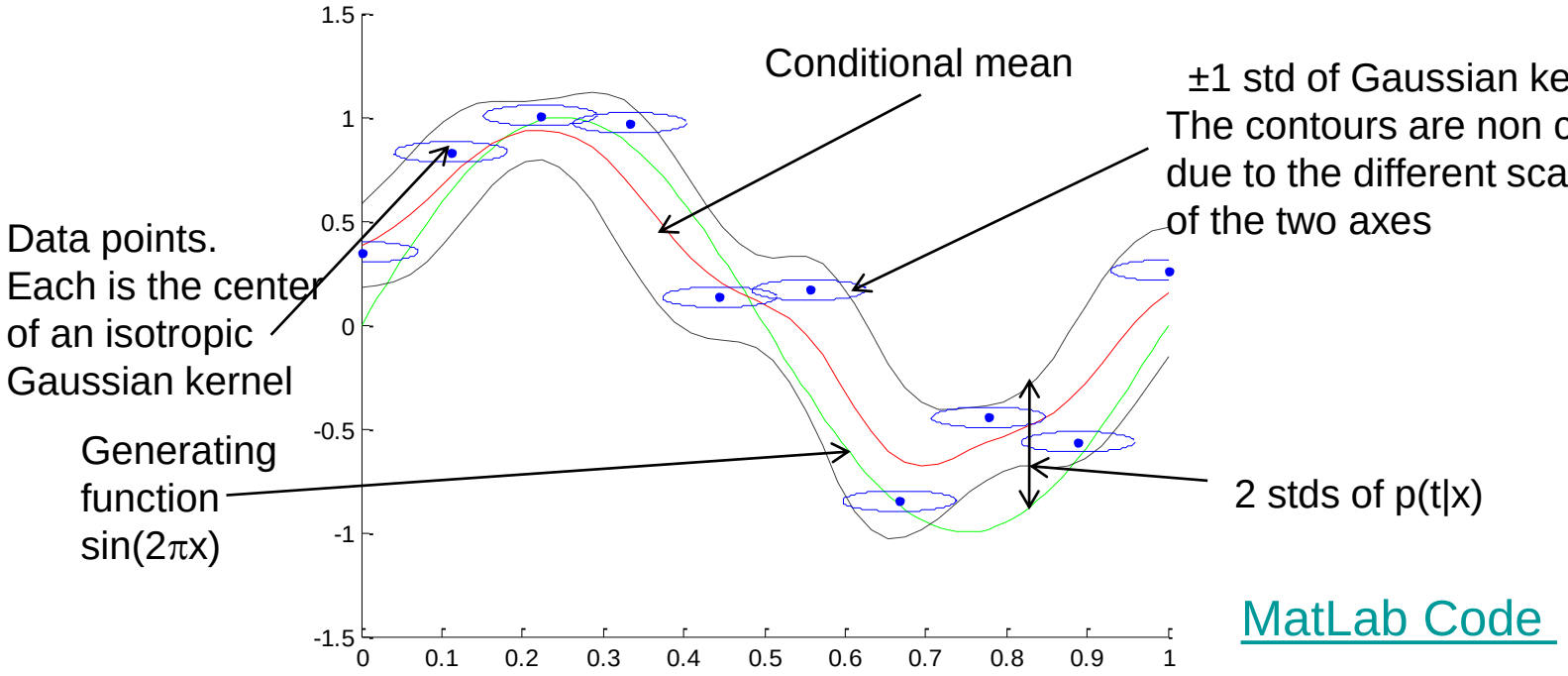
- In addition to the conditional expectation $y(\boldsymbol{x})$, the model provides the full conditional distribution:

$$p(t \mid \boldsymbol{x}) = \frac{p(t, \boldsymbol{x})}{\int p(t, \boldsymbol{x}) dt} = \frac{\sum_n f(\boldsymbol{x} - \boldsymbol{x}_n, t - t_n)}{\sum_m \int f(\boldsymbol{x} - \boldsymbol{x}_m, t - t_m) dt}$$

- For Gaussian component distributions, this is a mixture of Gaussians.

Kernel Regression: Nadaraya-Watson Model

- Single input variable x. $f(x,t)$ is a zero-mean isotropic Gaussian kernels over the variable $z=(x,t)$ with variance σ^2



MatLab Code

- Nadaraya, ´ E. A. (1964). On estimating regression. *Theory of Probability and its Applications* **9**(1), 141–142.
- Watson, G. S. (1964). Smooth regression analysis. *Sankhya: The Indian Journal of Statistics. Series A* **26**, 359–372.

Kernel Regression

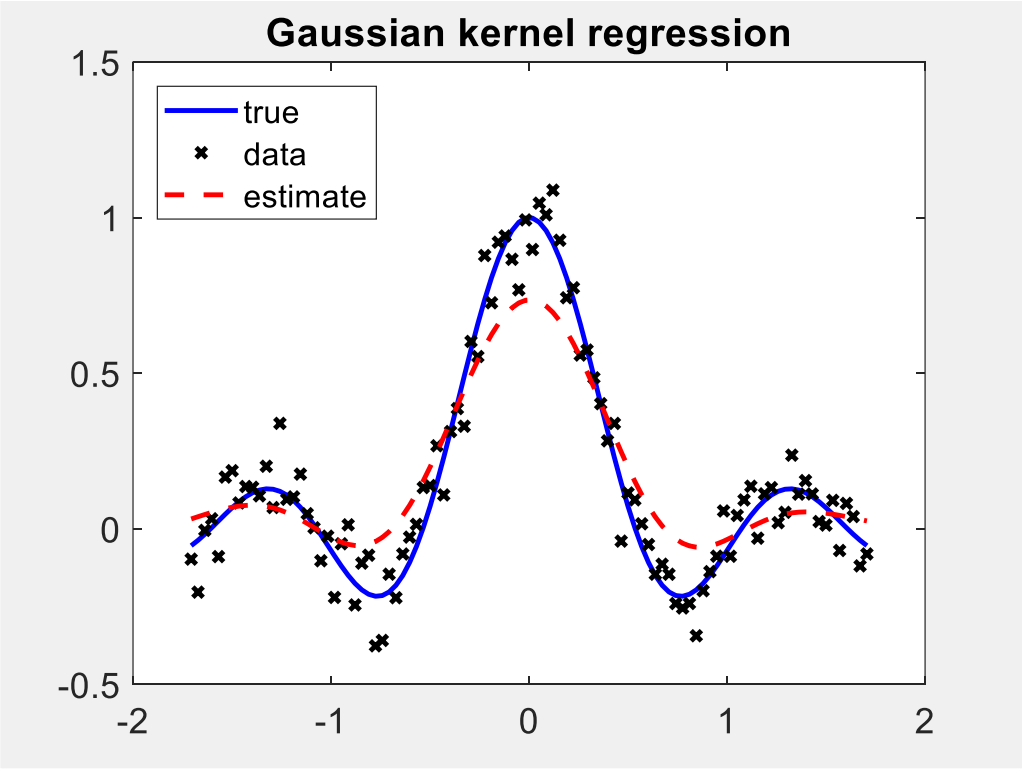
- This method only has one free parameter, namely h. For 1d data, if the true density is Gaussian and we are using Gaussian kernels, the optimal h is given by

$$h = \left(\frac{4}{3N}\right)^{1/5} \hat{\sigma}, \hat{\sigma} = 1.4826\ MAD, MAD\ = median(|x - median(x)|)$$

- We also take

$$h_x = h_t (= \sqrt{h_x h_t})$$

kernelRegressionDemo.m
from PMTK3



Locally Weighted Regression

- If we define $k_h(\boldsymbol{x} - \boldsymbol{x}_i) = k(\boldsymbol{x}, \boldsymbol{x}_i)$, we can rewrite the prediction made by kernel regression as

$$\hat{f}(\boldsymbol{x}_*) = \sum_{i=1}^{N} y_i \frac{k(\boldsymbol{x}_*, \boldsymbol{x}_i)}{\sum_{i'=1}^{N} k(\boldsymbol{x}_*, \boldsymbol{x}_{i'})}$$

- If $k(\boldsymbol{x}, \boldsymbol{x}_i)$ is not smoothing kernel we can write:

$$\hat{f}(\boldsymbol{x}_*) = \sum_{i=1}^{N} y_i \; k(\boldsymbol{x}_*, \boldsymbol{x}_i)$$

- This model is essentially fitting a constant function locally. We can fit a linear regression model for each $\boldsymbol{x}_*$ by solving

$$\max_{\beta(\boldsymbol{x}_*)} \sum_{i=1}^{N} k(\boldsymbol{x}_*, \boldsymbol{x}_i)(y_i - \boldsymbol{\beta}(\boldsymbol{x}_*)^T \boldsymbol{\phi}(\boldsymbol{x}_i))^2, \; \boldsymbol{\phi}(\boldsymbol{x}_i) = [1, x]$$

- This leads to a weighted LS problem $\boldsymbol{\beta}(\boldsymbol{x}_*) = (\boldsymbol{\Phi}^T \boldsymbol{D}(\boldsymbol{x}_*)\boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{D}(\boldsymbol{x}_*)\boldsymbol{y}$
  $D = diag\big(k(\boldsymbol{x}_*, \boldsymbol{x}_i)\big), \boldsymbol{\Phi}$ is an $N \times (D+1)$ design matrix
- Predictions take the form:

$$\hat{f}(\boldsymbol{x}_*) = \boldsymbol{\phi}(\boldsymbol{x}_*)^T \boldsymbol{\beta}(\boldsymbol{x}_*) = \boldsymbol{\phi}(\boldsymbol{x}_*)^T (\boldsymbol{\Phi}^T \boldsymbol{D}(\boldsymbol{x}_*)\boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{D}(\boldsymbol{x}_*)\boldsymbol{y} = \sum_{i=1}^{N} w_i(\boldsymbol{x}_*)\boldsymbol{y}_i$$

- Cleveland, W. and S. Devlin (1988). Locally-weighted regression: An approach to regression analysis by local fitting. *J. of the Am. Stat. Assoc. 83*(403), 596–610.
- Edakunni, N., S. Schaal, and S. Vijayakumar (2010). Probabilistic incremental locally weighted learning using randomly varying coefficient model. Technical report, USC.

Kernel Regression: Nadaraya-Watson Model

- We can use more flexible forms of Gaussian components, e.g. having different variance parameters for the input and target variables.

- We could alternatively model $p(t, \boldsymbol{x})$ using a Gaussian mixture model, trained using EM techniques and then find $p(t|\boldsymbol{x})$.

  - ✓ In this case we don't have a representation in terms of kernel functions evaluated at the training set data points.

  - ✓ However, the number of components in the mixture model can be smaller than the number of training set points, resulting in a model that is faster to evaluate for test data points.

- Ghahramani, Z. and M. I. Jordan (1994). Supervised learning from incomplete data via an EM appproach. In J. D. Cowan, G. T. Tesauro, and J. Alspector (Eds.), *Advances in Neural Information Processing Systems*, Volume 6, pp. 120–127.

The Kernel Trick

- Kernel Trick: We can work with the feature vectors $\boldsymbol{x}$, but modify the algorithm so that it replaces all inner products of the form $\langle \mathbf{x}, \mathbf{x}' \rangle$ with a call to the (Mercer) kernel function, $k(\boldsymbol{x}, \boldsymbol{x}')$.

- **Kernelized nearest neighbor classification:** Here we need to compute the Euclidean distance of a test vector to all the training points, find the closest one, and look up its label. This can be kernelized by observing that

$$||\boldsymbol{x}_i - \boldsymbol{x}_{i'}|| = \langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle + \langle \boldsymbol{x}_{i'}, \boldsymbol{x}_{i'} \rangle - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_{i'} \rangle$$

- This allows us to apply the nearest neighbor classifier to structured data objects.

Kernelized K-Medoids Clustering

- The K-medoids algorithm is similar to K-means - instead of representing each cluster's centroid by the mean of all data vectors assigned to the cluster, we make each centroid to be one of the data vectors themselves.

- Thus we always deal with integer indexes, rather than data objects.

- We assign objects to their closest centroids as before. When we update the centroids, we look at each point that belongs to the cluster, and measure the sum of its distances to all the others in the same cluster; we then pick the one which has the smallest such sum:

$$m_k = \underset{i:z_i=k}{\operatorname{argmin}} \sum_{i':z_{i'}=k} d(i,i'),\ d(i,i') = \|\boldsymbol{x}_i - \boldsymbol{x}_{i'}\|_2^2$$

Kernelized K-Medoids Clustering

- This takes $\mathcal{O}(n_k^2)$ work per cluster, whereas K-means takes $\mathcal{O}(n_k D)$ to update each cluster.

- This method can be modified to derive a classifier, by computing the nearest medoid for each class. This is known as nearest medoid classification. It can be kernelized by using

$$||\boldsymbol{x}_i - \boldsymbol{x}_{i'}|| = \langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle + \langle \boldsymbol{x}_{i'}, \boldsymbol{x}_{i'} \rangle - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_{i'} \rangle$$

- **1** *initialize* $m_{1:K}$ as a random subset of size K from $\{1, \ldots, N\}$;
- **2 repeat**
- **3** $z_i = \underset{k}{\operatorname{argmin}}\, d(i, m_k)$ for $i = 1 : N$;
- **4** $m_k \leftarrow \underset{i:z_i=k}{\operatorname{argmin}} \sum_{i':z_{i'}=k} d(i, i')$, for $k = 1 : K$;
- **5 until** *converged*;

- Hastie, T., R. Tibshirani, and J. Friedman (2009). *The Elements of Statistical Learning*. Springer. 2nd edition

Kernel PCA

Kernel PCA

- Kernel trick: take an algorithm expressed in terms of scalar products of the form $\boldsymbol{x}^T\boldsymbol{x}$ and generalize it by replacing the scalar products with a nonlinear kernel.

- Apply this technique of kernel substitution to PCA, obtaining a nonlinear algorithm called kernel PCA (Scholkopf et al. 1998)

- Consider a data set $\{\boldsymbol{x}_n\}, n = 1,\ldots,N$, of dimension D. Assume that we have already subtracted the sample mean from each $\boldsymbol{x}_n$, so that $\sum_n \boldsymbol{x}_n = 0$.

- Start by expressing PCA in such a form that the data $\{\boldsymbol{x}_n\}$ appear only in the form of the scalar products $\boldsymbol{x}_n^T\boldsymbol{x}_m$.

- Scholkopf, B., A. Smola, and K.-R. Muller (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation* **10**(5), 1299–1319.

Kernel PCA

- Recall that the principal components are defined by the normalized eigenvectors $\boldsymbol{u}_i$ of the $N \times N$ covariance matrix

$$\boldsymbol{S}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i, \text{ where: } \boldsymbol{S} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^T$$

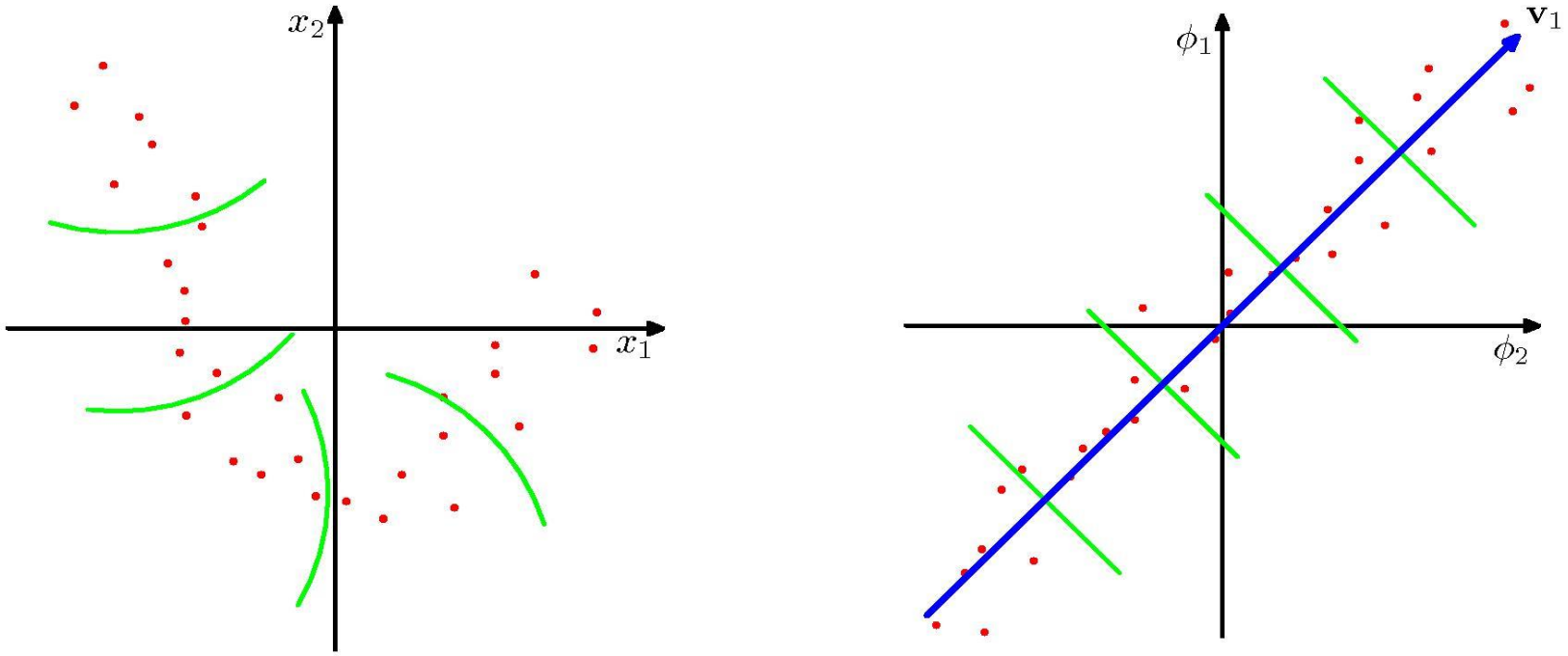
- Consider a nonlinear transformation $\boldsymbol{\phi}(\boldsymbol{x})$ into an M-dimensional feature space, so that each data point $\boldsymbol{x}_n$ is thereby projected onto a point $\boldsymbol{\phi}(\boldsymbol{x}_n)$.

- Now perform standard PCA in the feature space.
  - This implicitly defines a nonlinear principal component model in the original data space.

Kernel PCA

- A data set in the original data space (left) is projected by $\boldsymbol{\phi}(\boldsymbol{x})$ into a feature space (right).

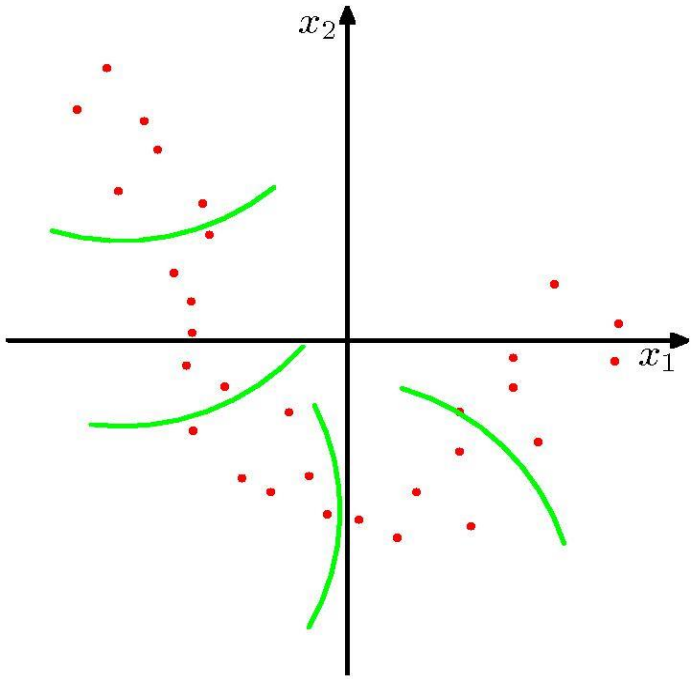
- By performing PCA in the feature space, we obtain the principal components ($\boldsymbol{v}_1$ being the 1st one).

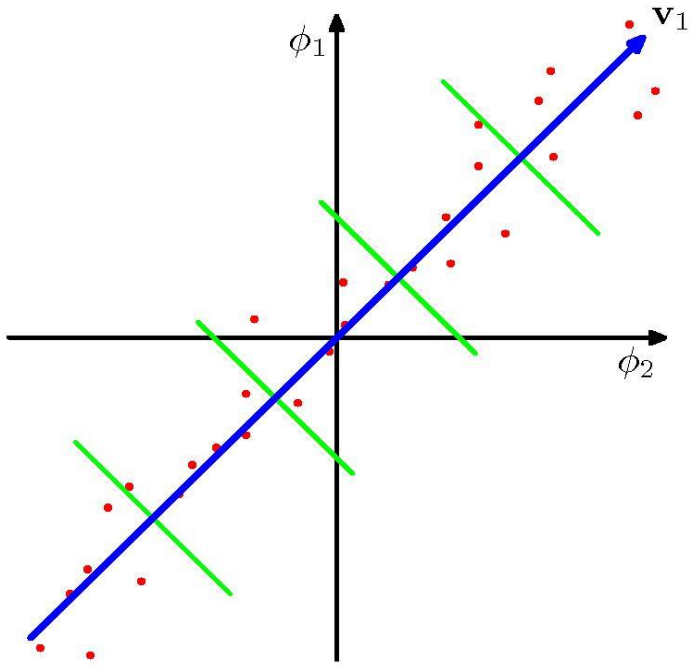


Kernel PCA

- The green lines in feature space indicate the linear projections onto $\boldsymbol{v}_1$, which correspond to nonlinear projections in the original data space.

- In general, it is not possible to represent the nonlinear principal component by a vector in $\boldsymbol{x}$ space.





kPCA

- Recall how we could compute a low-dimensional linear embedding of some data using PCA.
- This required finding the eigenvectors of the sample covariance matrix $\boldsymbol{S} = \frac{1}{N}\boldsymbol{X}^T\boldsymbol{X}$.
- However, we can also compute PCA by finding the eigenvectors of the inner product matrix $\boldsymbol{X}\boldsymbol{X}^T$, as we show below. This will allow us to produce a nonlinear embedding, using the kernel trick.
- First, let $\boldsymbol{U}$ be an orthogonal matrix containing the eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$ with corresponding eigenvalues in $\boldsymbol{\Lambda}$. By definition we have $(\boldsymbol{X}\boldsymbol{X}^T)\boldsymbol{U} = \boldsymbol{U}\boldsymbol{\Lambda}$. Pre-multiplying by $\boldsymbol{X}^T$ gives

N*N

$$(\boldsymbol{X}^T\boldsymbol{X})(\boldsymbol{X}^T\boldsymbol{U}) = (\boldsymbol{X}^T\boldsymbol{U})\boldsymbol{\Lambda}$$

  from which we see that the eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$ (and hence of $\boldsymbol{S}$) are $\boldsymbol{V} = \boldsymbol{X}^T\boldsymbol{U}$, with eigenvalues given by $\boldsymbol{\Lambda}$ as before. However, these eigenvectors are not normalized, since

$$||\boldsymbol{v}_j||^2 = \boldsymbol{u}_j^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{u}_j = \lambda_j\boldsymbol{u}_j^T\boldsymbol{u}_j = \lambda_j.$$

- So the normalized eigenvectors are given by $\boldsymbol{V}_{pca} = \boldsymbol{X}^T\boldsymbol{U}\boldsymbol{\Lambda}^{-1/2}$.

kPCA

- This is a useful trick for PCA if $D > N$, since $\boldsymbol{X}^T\boldsymbol{X}$ has size $D \times D$, whereas $\boldsymbol{X}\boldsymbol{X}^T$ has size $N \times N$. It also allow us to use the kernel trick.

- Let $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^T$ be the Gram matrix. From Mercer's theorem the use of a kernel implies some underlying feature space, so we are implicitly replacing $\boldsymbol{x}_i$ with $\boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{\phi}_i$. Let $\boldsymbol{\Phi}$ be the corresponding design matrix, and $\boldsymbol{S}_\phi = \frac{1}{N}\sum_i \boldsymbol{\phi}_i \boldsymbol{\phi}_i^T$ be the corresponding covariance matrix in feature space. The eigenvectors are given by $\boldsymbol{V}_{kPCA} = \boldsymbol{\Phi}^T\boldsymbol{U}\boldsymbol{\Lambda}^{-1/2}$, where $\boldsymbol{U}, \boldsymbol{\Lambda}$ contain the eigenvectors & eigenvalues of $\boldsymbol{K}$.

- We cannot compute $\boldsymbol{V}_{kPCA}$, since $\boldsymbol{\phi}_i$ is potentially infinite dimensional. However, we can compute the projection of a test vector $\boldsymbol{x}_*$ onto the feature space as follows:

$$\boldsymbol{\phi}_*^T\boldsymbol{V}_{kPCA} = \boldsymbol{\phi}_*^T\boldsymbol{\Phi}^T\mathbf{U}\boldsymbol{\Lambda}^{-1/2} = \boldsymbol{k}_*^T\boldsymbol{U}\boldsymbol{\Lambda}^{-1/2}$$

  where $\boldsymbol{k}_* = [\kappa(\boldsymbol{x}_*, \boldsymbol{x}_1), \ldots, k(\boldsymbol{x}_*, \boldsymbol{x}_N)]$.

- So far, we have assumed the projected data has zero mean, which is not the case in general.

Kernel PCA

- For the moment, let us assume that the projected data set also has zero mean, so that $\sum_{n=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_n) = 0$. We shall return to this point shortly.

- The $D_* \times D_*$ sample covariance matrix in feature space is given by

$$C = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_n) \boldsymbol{\phi}(\boldsymbol{x}_n)^T$$

- The eigenvector expansion is defined as:

$$C\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i, \; i = 1, \ldots, D_*$$

- Can we solve this eigenvalue problem without having to work explicitly in the feature space?

Kernel PCA

$$C = \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\boldsymbol{\phi}(\boldsymbol{x}_n)^T \qquad C\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i, i = 1,\ldots,D_*$$

- From the definition of $\boldsymbol{C}$, the eigenvector equations tells us that $\boldsymbol{v}_i$ satisfies

$$\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\left\{\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{v}_i\right\} = \lambda_i \boldsymbol{v}_i, i = 1,\ldots,D_*$$

- We see that (provided $\lambda_i > 0$), the vector v_i is given by a linear combination of the $\boldsymbol{\phi}(\boldsymbol{x}_n)$ and so can be written in the form

$$\boldsymbol{v}_i = \sum_{n=1}^{N} a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n), i = 1,\ldots,D_*$$

Kernel PCA

$$\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\left\{\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{v}_i\right\}=\lambda_i\boldsymbol{v}_i,\ i=1,\ldots,D_*$$

- Substituting

$$\boldsymbol{v}_i=\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n),\ i=1,\ldots,D_*$$

back into the eigenvector equation, we obtain

$$\left[\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\boldsymbol{\phi}(\boldsymbol{x}_n)^T\right]\sum_{m=1}^{N}a_{im}\boldsymbol{\phi}(\boldsymbol{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n),\ i=1,\ldots,D_*$$

- The key step is now to express this in terms of the kernel function

$$K_{nm}\equiv k(\boldsymbol{x}_n,\boldsymbol{x}_m)=\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{\phi}(\boldsymbol{x}_m)$$

which we do by multiplying both sides by $\boldsymbol{\phi}(\boldsymbol{x}_l)^T$.

Kernel PCA

$$\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_n)\boldsymbol{\phi}(\boldsymbol{x}_n)^T\sum_{m=1}^{N}a_{im}\boldsymbol{\phi}(\boldsymbol{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n),\ i=1,\ldots,D_*$$

- Multiplying both sides by $\boldsymbol{\phi}(\boldsymbol{x}_l)^T$ to give

$$\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_l)^T\boldsymbol{\phi}(\boldsymbol{x}_n)\boldsymbol{\phi}(\boldsymbol{x}_n)^T\sum_{m=1}^{N}a_{im}\boldsymbol{\phi}(\boldsymbol{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\boldsymbol{x}_l)^T\boldsymbol{\phi}(\boldsymbol{x}_n),\ i=1,\ldots,D_*\Rightarrow$$

$$\frac{1}{N}\sum_{n=1}^{N}k(\boldsymbol{x}_l,\boldsymbol{x}_n)\sum_{m=1}^{N}a_{im}k(\boldsymbol{x}_n,\boldsymbol{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}k(\boldsymbol{x}_l,\boldsymbol{x}_n),\ i=1,\ldots,D_*\Rightarrow$$

- This can be written in a matrix form as:

$$\boldsymbol{K}^2\boldsymbol{a}_i=\lambda_i N\boldsymbol{K}\boldsymbol{a}_i$$

$\boldsymbol{a}_i$ N-dimensional column vector with elements $a_{in}, n = 1,..,N$.

- Can simplify $\boldsymbol{K}\boldsymbol{a}_i=\lambda_i N\boldsymbol{a}_i$. These Eqs. differ by eigen-vectors of $\boldsymbol{K}$ with 0 eigenvalues that don't affect the principal components projection.

Kernel PCA

$$\boldsymbol{K}\boldsymbol{a}_i = \lambda_i N \boldsymbol{a}_i$$

- We can find solutions for $\boldsymbol{a}_i$ by solving the above eigenvalue problem.
- The normalization condition for the coefficients $\boldsymbol{a}_i$ is obtained by requiring that the eigenvectors in feature space be normalized.
- Using

and

$$\boldsymbol{v}_i = \sum_{n=1}^{N} a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n),\ i = 1,\ldots,D_*$$

we have:

$$\boldsymbol{K}\boldsymbol{a}_i = \lambda_i N \boldsymbol{a}_i$$

$$1 = \boldsymbol{v}_i^T \boldsymbol{v}_i = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{N} a_{in}a_{im}\boldsymbol{\phi}(\boldsymbol{x}_n)^T\boldsymbol{\phi}(\boldsymbol{x}_m) = \boldsymbol{a}_i^T\boldsymbol{K}\boldsymbol{a}_i = \lambda_i N \boldsymbol{a}_i^T\boldsymbol{a}_i \Rightarrow \boxed{1 = \lambda_i N \boldsymbol{a}_i^T\boldsymbol{a}_i}$$

Kernel PCA

- The principal component projections can now be cast in terms of the kernel function.

- Using

$$\boldsymbol{v}_i = \sum_{n=1}^{N} a_{in} \boldsymbol{\phi}(\boldsymbol{x}_n), i = 1, \ldots, D_*$$

the projection of a point $\boldsymbol{x}$ onto eigenvector i is given by

$$y_i(\boldsymbol{x}) = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{v}_i = \sum_{n=1}^{N} a_{in} \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}_n) = \sum_{n=1}^{N} a_{in} k(\boldsymbol{x}, \boldsymbol{x}_n)$$

Kernel PCA

- In the original D-dimensional $\boldsymbol{x}$ space: D orthogonal eigenvectors and thus at most D linear principal components.
- The dimensionality D_* of the feature space, however, can be much larger than D (even infinite), and thus we can find a number of nonlinear principal components that can exceed D.
- Note, however, that the number of nonzero eigenvalues cannot exceed the number N of data points:
  - The covariance matrix in feature space has rank at most equal to N.
  - Recall that kernel PCA involves the eigenvector expansion of the $N \times N$ matrix $\boldsymbol{K}$.

Kernel PCA

- So far we have assumed that the projected data set given by $\boldsymbol{\phi}(\boldsymbol{x}_n)$ has zero mean, which in general will not be the case.

- We cannot simply compute and then subtract off the mean, since we want to formulate the algorithm purely in terms of the kernel function.

- The projected data points after centralizing are given by

$$\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_n) = \boldsymbol{\phi}(\boldsymbol{x}_n) - \frac{1}{N}\sum_{l=1}^{N}\boldsymbol{\phi}(\boldsymbol{x}_l)$$

- The corresponding elements of the Gram matrix $\tilde{\boldsymbol{K}}_{nm} = \tilde{\boldsymbol{\phi}}(x_n)^T\tilde{\boldsymbol{\phi}}(x_m)$ can be computed as shown next.

Kernel PCA

$$\tilde{\boldsymbol{K}}_{nm} = \tilde{\boldsymbol{\phi}}(x_n)^T \tilde{\boldsymbol{\phi}}(x_m) = \left(\boldsymbol{\phi}(\boldsymbol{x}_n) - \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_l) \right)^T \left(\boldsymbol{\phi}(\boldsymbol{x}_m) - \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_l) \right)$$

$$= \boldsymbol{\phi}(\boldsymbol{x}_n)^T \boldsymbol{\phi}(\boldsymbol{x}_m) - \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_n)^T \boldsymbol{\phi}(\boldsymbol{x}_l)$$

$$- \frac{1}{N}\sum_{l=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_l)^T \boldsymbol{\phi}(\boldsymbol{x}_m) + \frac{1}{N^2}\sum_{j=1}^{N}\sum_{l=1}^{N} \boldsymbol{\phi}(\boldsymbol{x}_j)^T \boldsymbol{\phi}(\boldsymbol{x}_l)$$

$$= k(\boldsymbol{x}_n, \boldsymbol{x}_m) - \frac{1}{N}\sum_{l=1}^{N} k(\boldsymbol{x}_n, \boldsymbol{x}_l) - \frac{1}{N}\sum_{l=1}^{N} k(\boldsymbol{x}_l, \boldsymbol{x}_m) + \frac{1}{N^2}\sum_{j=1}^{N}\sum_{l=1}^{N} k(\boldsymbol{x}_j, \boldsymbol{x}_l)$$

- We can express this in matrix notation as:

$$\tilde{\boldsymbol{K}} = \boldsymbol{K} - \mathbf{1}_N \boldsymbol{K} - \boldsymbol{K}\mathbf{1}_N + \mathbf{1}_N \boldsymbol{K} \mathbf{1}_N$$

where $\mathbf{1}_N$ denotes the $N \times N$ matrix in which every element takes the value $1/N$.

Kernel PCA

$$\widetilde{\boldsymbol{K}} = \boldsymbol{K} - \mathbf{1}_N \boldsymbol{K} - \boldsymbol{K}\mathbf{1}_N + \mathbf{1}_N \boldsymbol{K} \mathbf{1}_N$$

- Thus we can evaluate $\widetilde{\boldsymbol{K}}$ using only the kernel function and then use $\widetilde{\boldsymbol{K}}$ to determine the eigenvalues and eigenvectors.
- Note that the standard PCA algorithm is recovered as a special case if we use a linear kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^T \boldsymbol{x}'$.
- We often use Gaussian kernels of the form:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|}{2\sigma^2}\right)$$

- Whereas linear PCA is limited to using $M \leq D$ components, in kPCA, we can use up to N components, since $\boldsymbol{\Phi}$ is $N \times D_*$, where D_* is the (potentially infinite) dimensionality of the embedded feature vectors.

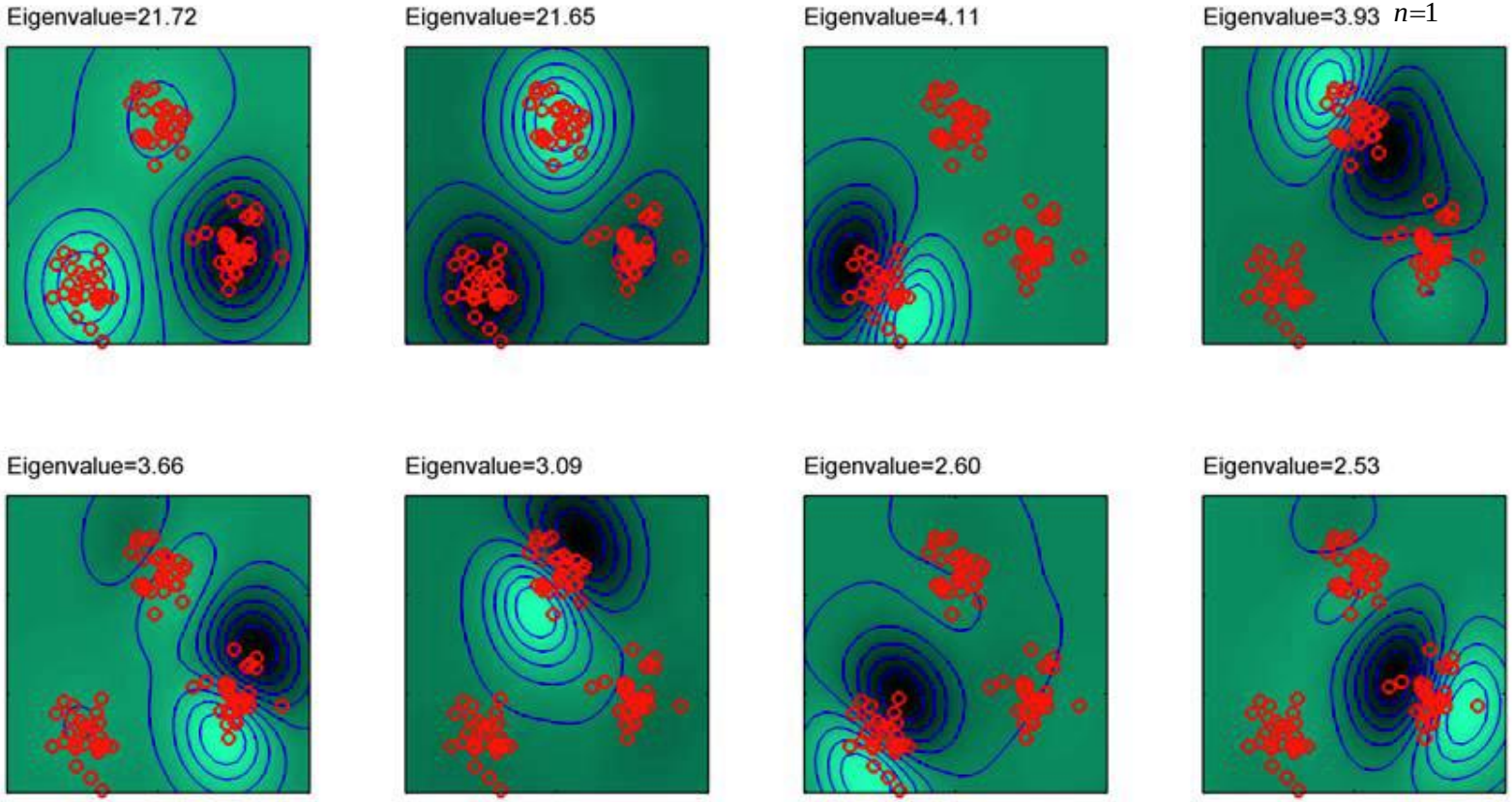
kPCA - Algorithm

- **1** Input: $\boldsymbol{K}$ of size $N \times N$, $\boldsymbol{K}_*$ of size $N_* \times N$, number of latent dimensions M;
- **2** $O = \frac{1}{N}\mathbf{1}_N\mathbf{1}_N^T$;
- **3** $\widetilde{\boldsymbol{K}} = \boldsymbol{K} - \boldsymbol{O}\boldsymbol{K} - \boldsymbol{K}\boldsymbol{O} + \boldsymbol{O}\boldsymbol{K}\boldsymbol{O}$;
- **4** $[\boldsymbol{U}, \boldsymbol{\Lambda}] = eig(\widetilde{\boldsymbol{K}})$;
- **5 for** $i = 1 : N$ **do**
- **6** $\quad \boldsymbol{v}_i = \boldsymbol{u}_i/\sqrt{\lambda_i}$;
- **7** $\boldsymbol{O}_* = \frac{1}{N}\mathbf{1}_{\boldsymbol{N}_*\boldsymbol{N}}\mathbf{1}_{\boldsymbol{N}_*}^T$;
- **8** $\widetilde{\boldsymbol{K}}_* = \boldsymbol{K}_* - \boldsymbol{O}_*\boldsymbol{K}_* - \boldsymbol{K}_*\boldsymbol{O}_* + \boldsymbol{O}_*\boldsymbol{K}_*\boldsymbol{O}_*$;
- **9** $\boldsymbol{Z} = \widetilde{\boldsymbol{K}}_*\boldsymbol{V}(:, 1 : M)$

Kernel PCA

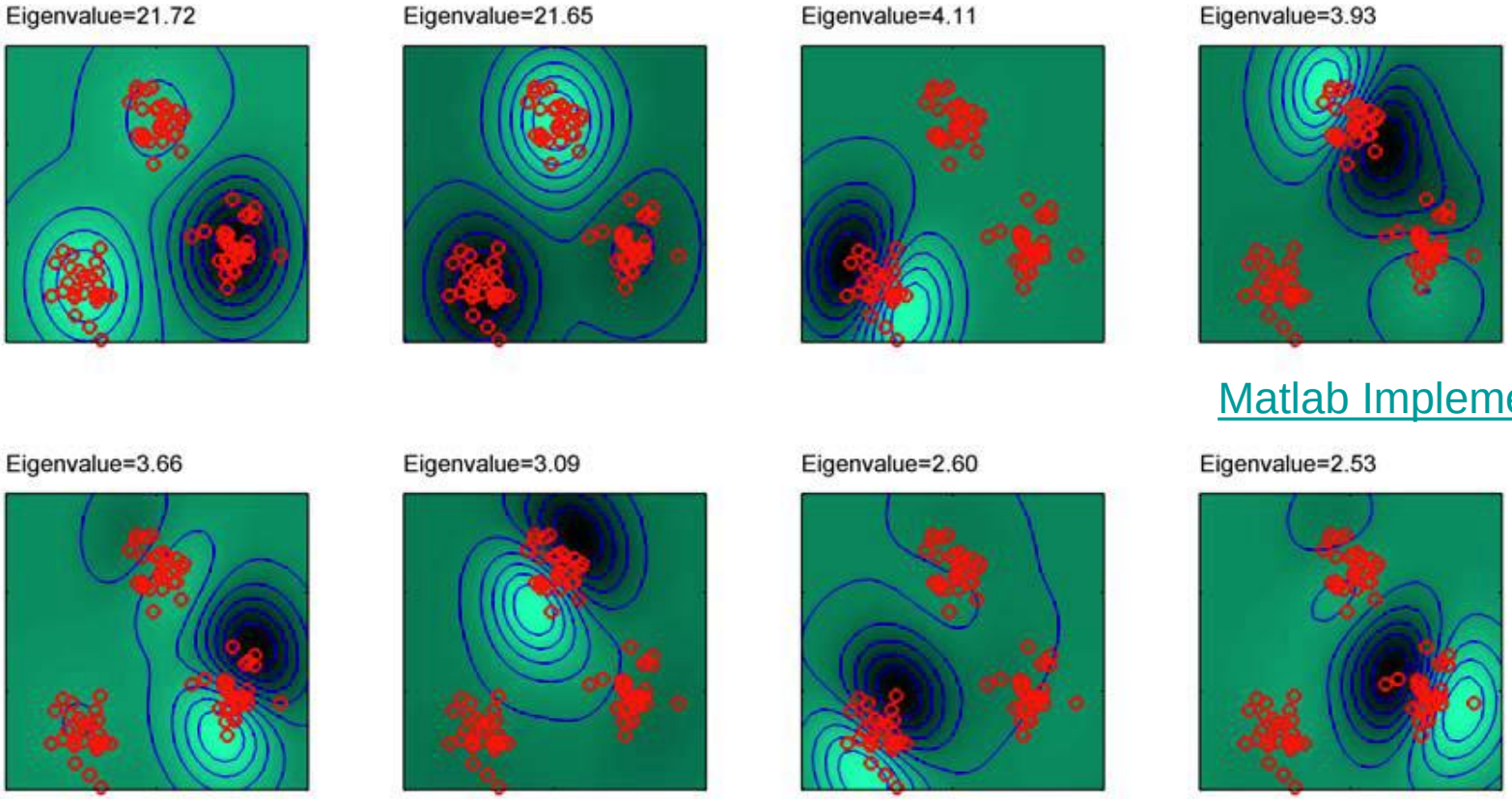
- KPCA, with a Gaussian kernel ($2\sigma^2 = 0.1$) applied to a synthetic data in 2D, shows the first 8 eigenfunctions and eigenvalues.
- The lines along which the projection onto the corresponding principal component is constant are shown:

$$\boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{v}_i = \sum_{n=1}^{N} a_{in} k(\boldsymbol{x}, \boldsymbol{x}_n)$$



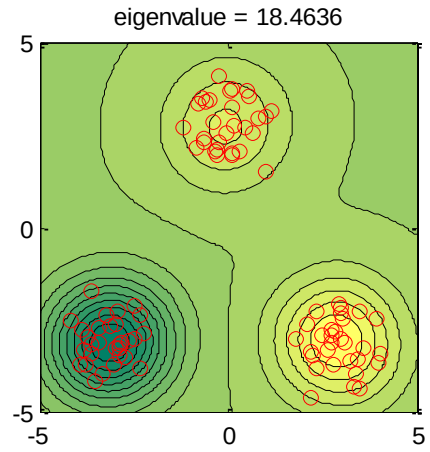
Kernel PCA

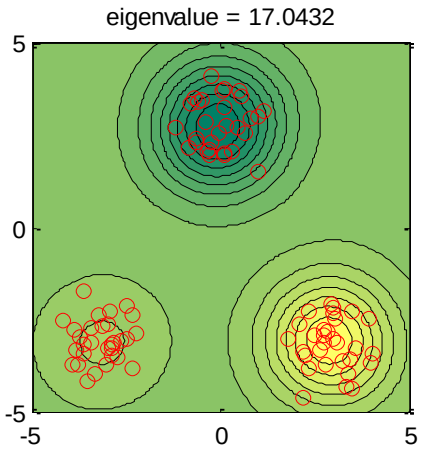
- Note the first 2 eigenvectors separate the 3 clusters, the next three eigenvectors split each of the cluster into halves, and the following three eigenvectors again split the clusters into halves along directions orthogonal to the previous splits.

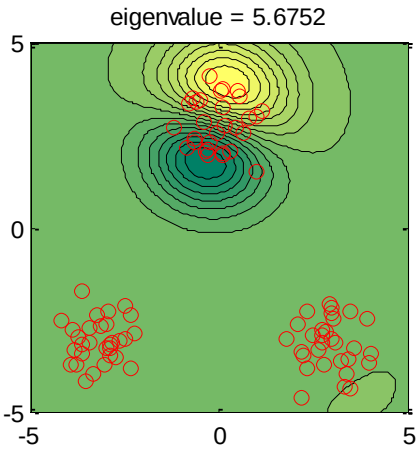


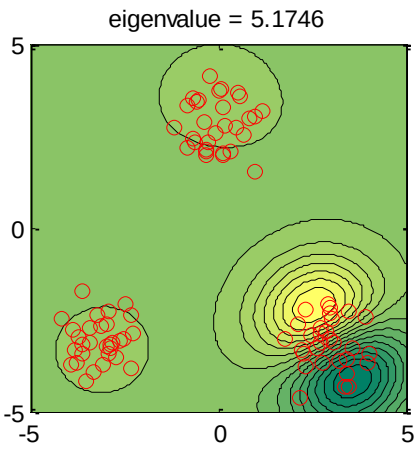
Matlab Implementation

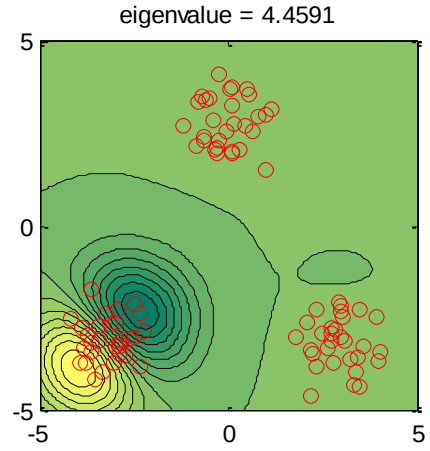
Kernel PCA

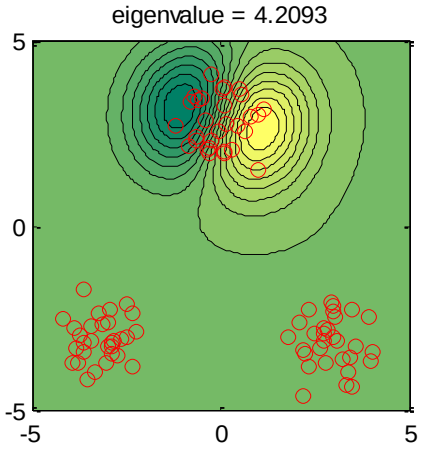


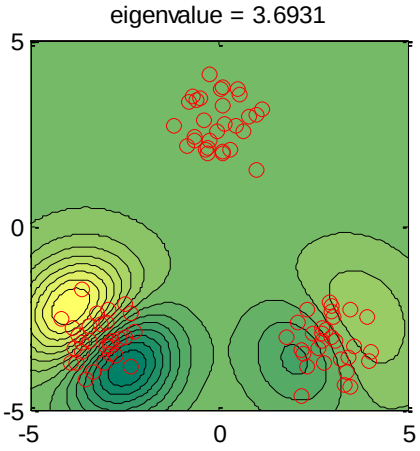


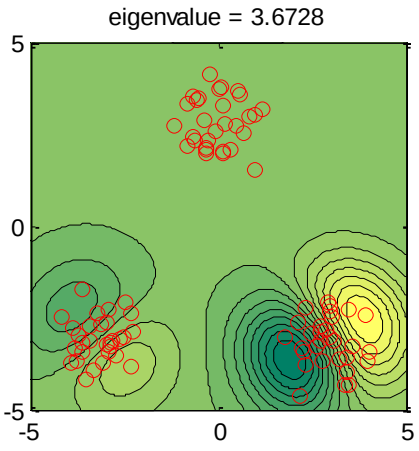








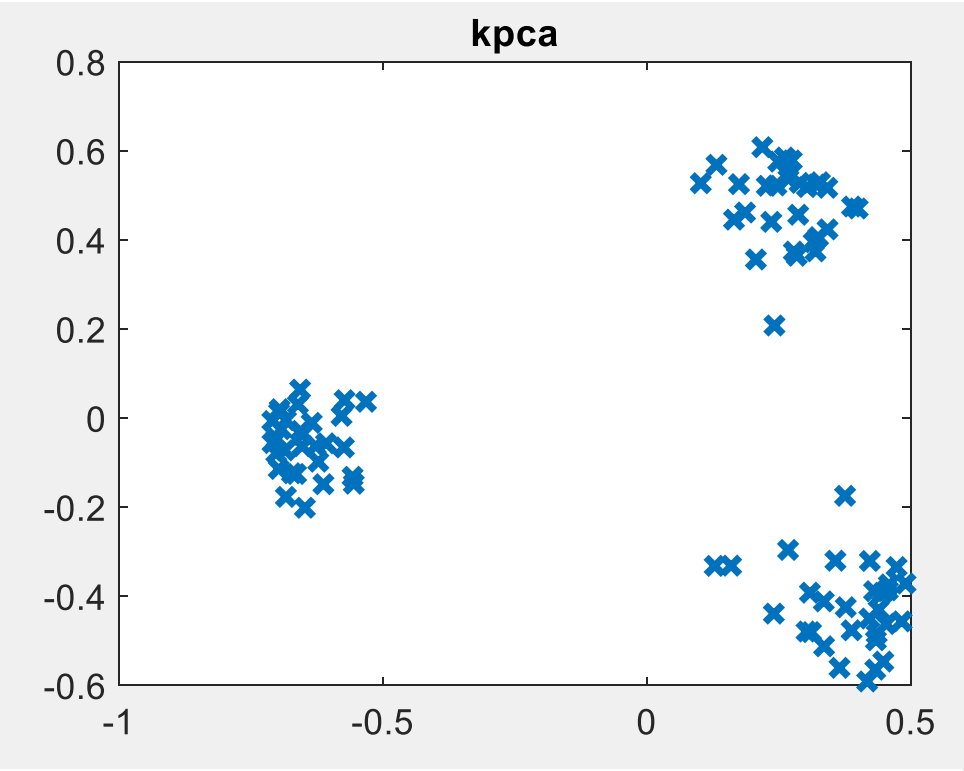


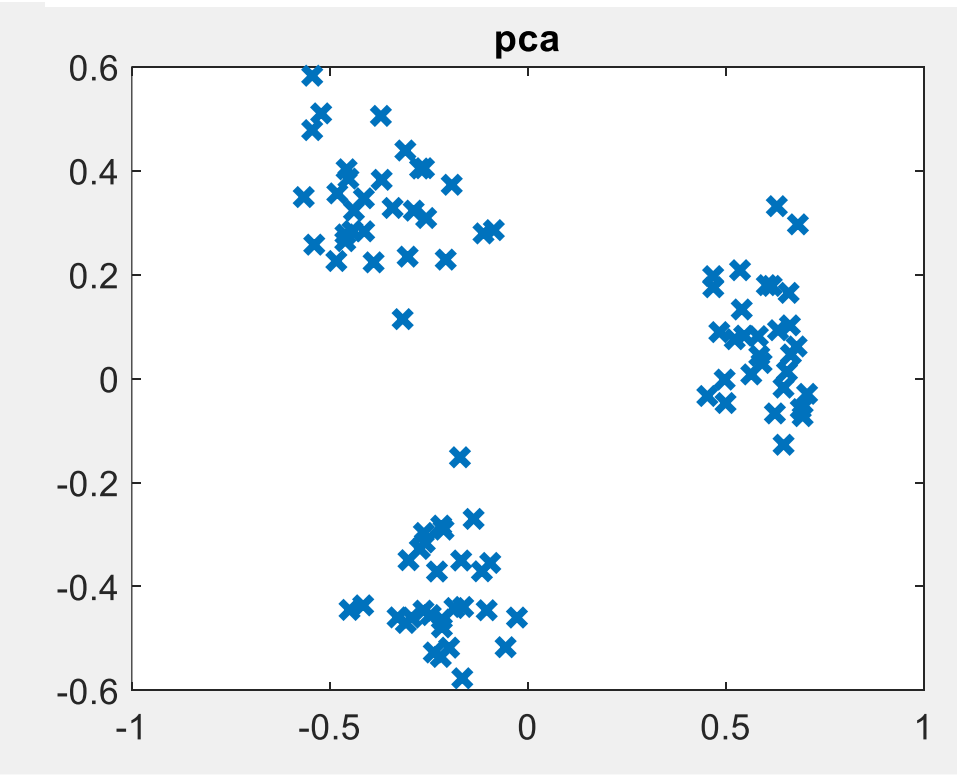


Matlab Implementation

Kernel PCA

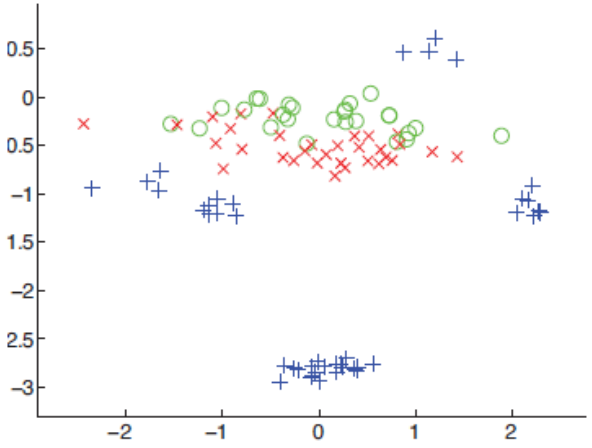
- Projection of the data from onto the first 2 principal bases computed using PCA and kPCA. PCA perfectly represents the data. kPCA represents each cluster by a different line.



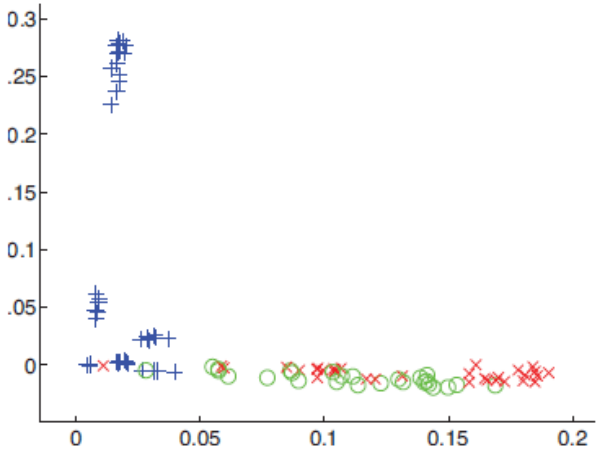


Oil Data Set

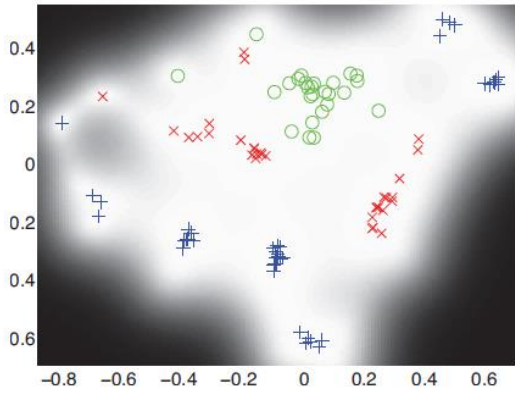
- We will use a 12 dimensional data set representing the three known phases of flow in an oil pipeline.
- We project this into 2d using PCA and kPCA (with an RBF kernel).
- If we perform nearest neighbor classification in the low-dimensional space, kPCA makes 13 errors and PCA makes 20.



PCA



kPCA



GPLVM with Gaussian kernel

- Lawrence, N. D. (2005). Probabilistic non-linear principal component analysis with gaussian process latent variable models. *J. of Machine Learning Research 6*, 1783–1816.

Kernel PCA: Disadvatages

- KPCA involves finding the eigenvectors of the $N \times N$ matrix $\widetilde{\boldsymbol{K}}$ rather than the $D \times D$ matrix $\boldsymbol{S}$ of PCA.
  - In practice, for large N approximations are used.
- In PCA, we retain some reduced number $M < D$ of eigenvectors and then approximate a data vector $\boldsymbol{x}_n$ by its projection $\hat{\boldsymbol{x}}_n$ onto the M-dimensional principal subspace, defined by

$$\hat{\boldsymbol{x}}_n = \sum_{i=1}^{M} (\boldsymbol{x}_n^T \boldsymbol{u}_i)\, \boldsymbol{u}_i$$

- In KPCA, this in general is not possible.

Kernel PCA: Preimage problem

$$\hat{\boldsymbol{x}}_n = \sum_{i=1}^{M} (\boldsymbol{x}_n^T \boldsymbol{u}_i)\, \boldsymbol{u}_i$$

- This is not possible in KPCA. Indeed note that the mapping $\boldsymbol{\phi}(\boldsymbol{x})$ maps the D-dimensional $\boldsymbol{x}$ space into a D-dimensional manifold in the D_*-dimensional feature space $\boldsymbol{\phi}$.

- The vector $\boldsymbol{x}$ is known as the pre-image of $\boldsymbol{\phi}(\boldsymbol{x})$. The projection of points in feature space onto the linear PCA subspace in that space will typically not lie on the nonlinear D-dimensional manifold and so will not have a corresponding pre-image in data space.

- Techniques have been proposed for finding approximate pre-images (Bakir et al., 2004)

  - Bakir, G. H., J. Weston, and B. Scholkopf (2004). Learning to find pre-images. In S. Thrun, L. K. Saul, and B. Scholkopf (Eds.), *Advances in Neural Information Processing Systems*, Volume 16, pp. 449–456. MIT Press.